Debugging ART Garbage Collection

This document describes how to debug Android Runtime (ART) Garbage Collection (GC) correctness and performance issues. It explains how to use GC verification options, identify solutions for GC verification failures, and measure and address GC performance problems.

See ART and Dalvik (https://source.android.com/devices/tech/dalvik/index.html), the Dalvik Executable format (https://source.android.com/devices/tech/dalvik/dex-format.html), and the remaining pages within this ART and Dalvik (https://source.android.com/devices/tech/dalvik/index.html) section to work with ART. See Verifying App Behavior on the Android Runtime (ART) (http://developer.android.com/guide/practices/verifying-apps-art.html) for additional help verifying app behavior.

ART GC overview

ART, introduced as a developer option in Android 4.4, is the default Android runtime for Android 5.0 and beyond. The Dalvik runtime is no longer maintained or available and its byte-code format is now used by ART. Please note this section merely summarizes ART's GC. For additional information, see the Android runtime (https://www.google.com/events/io/io14videos/b750c8da-aebe-e311-b297-00155d5066d7) presentation conducted at Google I/O 2014.

ART has a few different GC plans that consist of running different garbage collectors. The default plan is the CMS (concurrent mark sweep) plan, which uses mostly sticky CMS and partial CMS. Sticky CMS is ART's non-moving generational garbage collector. It scans only the portion of the heap that was modified since the last GC and can reclaim only the objects allocated since the last GC. In addition to the CMS plan, ART performs heap compaction when an app changes process state to a jank-imperceptible process state (e.g. background or cached).

Aside from the new garbage collectors, ART also introduces a new bitmap-based memory allocator called RosAlloc (runs of slots allocator). This new allocator has sharded locking and outperforms DlMalloc by adding thread local buffers for small allocation sizes.

Compared to Dalvik, the ART CMS garbage collection plan has a number of improvements:

- The number of pauses is reduced from two to one compared to Dalvik. Dalvik's first pause, which did mostly root marking, is done concurrently in ART by getting the threads to mark their own roots, then resume running right away.
- Similarly to Dalvik, the ART GC also has a pause before the sweeping process. The key difference in this area is that some of the Dalvik phases during this pause are done concurrently in ART. These phases include `java.lang.ref.Reference` processing, system weak sweeping (e.g. jni weak globals, etc.), re-marking non-thread roots, and card pre-cleaning. The phases that are still done paused in ART are scanning the dirty cards and re-marking the thread roots, which helps reduce the pause time.
- The last area where the ART GC improves over Dalvik is with increased GC throughput enabled by the sticky CMS collector. Unlike normal generational GC, sticky CMS is non-moving. Instead of having a dedicated region for young objects, young objects are kept in an allocation stack, which is basically an array of `java.lang.Object`. This avoids moving objects required to maintain low pauses but has the disadvantage of having longer collections for heaps with complex object graphs.

The other main area where the ART GC is different than Dalvik is the introduction of moving garbage collectors. The goal of moving GCs is to reduce memory usage of backgrounded apps through heap compaction. Currently, the event that triggers heap compaction is ActivityManager process-state changes. When an app goes to background, it notifies ART the process state is no longer jank "perceptible." This enables ART do things that cause long application thread pauses, such as compaction and monitor deflation. The two current moving GCs that are in use are homogeneous space compaction and semi-space compaction.

- Semi-space compaction moves objects between two tightly packed bump pointer spaces. This moving GC occurs on low-memory devices since it saves slightly more memory than homogeneous space compaction. The additional savings come mostly from having tightly packed objects, which avoid RosAlloc / DlMalloc allocator accounting overhead. Since CMS is still used in the foreground and it can't collect from a bump pointer space, semi space requires another transition when the app is foregrounded. This is not ideal since it can cause a noticeable pause.
- Homogenous space compaction works by copying from one RosAlloc space to another RosAlloc space. This helps reduce memory usage by reducing heap fragmentation. This is currently the default compaction mode for non-low-memory devices. The main advantage that homogeneous space compaction has over semi-space compaction is not needing a heap transition when the app goes back to foreground.

GC verification and performance options

It is possible to change the GC type if you are an OEM. The process for doing this involves modifying system properties through adb. Keep in mind that these are only modifiable on non-user or rooted builds.

Changing the GC type

There are ways to change the GC plans that ART uses. The main way to change the foreground GC plan is by changing the **`dalvik.vm.gctype`** property or passing in an **`-Xgc:`** option. It is possible to pass in multiple GC options separated by commas.

In order to derive the entire list of available **`-Xgc`** settings, it is possible to type **`adb shell dalvikvm -help`** to print the various runtime command-line options.

Here is one example that changes the GC to semi space and turns on pre-GC heap verification: **`adb shell setprop dalvik.vm.gctype SS,preverify`**

- **`CMS`**, which is also the default value, specifies the concurrent mark sweep GC plan. This plan consists of running sticky generational CMS, partial CMS, and full CMS. The allocator for this plan is RosAlloc for movable objects and DlMalloc for non-movable objects.
- **`SS`** specifies the semi space GC plan. This plan has two semi spaces for movable objects and a DlMalloc space for non-movable objects. The movable object allocator defaults to a shared bump pointer allocator which uses atomic operations. However, if the **`-XX:UseTLAB`** flag is also passed in, the allocator uses thread local bump pointer allocation.
- **`GSS`** specifies the generational semi space plan. This plan is very similar to the semi-space plan with the exception that older-lived objects are promoted into a large RosAlloc space. This has the advantage of needing to copy significantly fewer objects for typical use cases.

Verifying the heap

Heap verification is probably the most useful GC option for debugging GC-related errors or heap corruption. Enabling heap verification causes the GC to check the correctness of the heap at a few points during the garbage collection process. Heap verification shares the same options as the ones that change the GC type. If enabled, heap verification verifies the roots and ensures that reachable objects reference only other reachable objects. GC verification is enabled by passing in the following -**`Xgc`** values:

- If enabled, **`[no]preverify`** performs heap verification before starting the GC.
- If enabled, **`[no]presweepingverify`** performs heap verification before starting the garbage collector sweeping process.
- If enabled, **`[no]postverify`** performs heap verification after the GC has finished sweeping.
- **`[no]preverify_rosalloc`**, **`[no]postsweepingverify_rosalloc`**, **`[no]postverify_rosalloc`** are also additional GC options that verify only the state of RosAlloc's internal accounting. The main things verified are that magic values match expected constants, and free blocks of memory are all registered in the **`free_page_runs_`** map.

Using the TLAB allocator option

Currently, the only option that changes the allocator used without affecting the active GC type is the TLAB option. This option is not available through system properties but can be enabled by passing in -**`XX:UseTLAB`** to **`dalvikvm`**. This option enables faster allocation by having a shorter allocation code path. Since this option requires using either the SS or GSS GC types, which have rather long pauses, it is not enabled by default.

Performance

There are two main tools that can be used to measure GC performance. GC timing dumps and systrace. The most visual way to measure GC performance problems would be to use systrace to determine which GCs are causing long pauses or preempting application threads. Although the ART GC is relatively efficient, there are still a few ways to get performance problems through excessive allocations or bad mutator behavior.

Ergonomics

Compared to Dalvik, ART has a few key differences regarding GC ergonomics. One of the major improvements compared to Dalvik is no longer having GC for alloc in cases where we start the concurrent GC later than needed. However, there is one downside to this, not blocking on the GC can cause the heap to grow more than Dalvik in some circumstances. Fortunately, ART has heap compaction, which mitigates this issue by defragmenting the heap when the process changes to a background process state.

The CMS GC ergonomics have two types of GCs that are regularly run. Ideally, the GC ergonomics will run the generational sticky CMS more often than the partial CMS. The GC does sticky CMS until the throughput (calculated by bytes freed / second of GC duration) of the last GC is less than the mean throughput of partial CMS. When this occurs, the ergonomics plan the next concurrent GC to be a partial CMS instead of sticky CMS. After the partial CMS completes, the ergonomics changes the next GC back to sticky CMS. One key factor that makes the ergonomics work is that sticky CMS doesn't adjust the heap footprint limit after it completes. This causes sticky CMS to happen more and more often until the throughput is lower than partial CMS, which ends up growing the heap.

Using SIGQUIT to obtain GC performance info

It is possible to get GC performance timings for apps by sending SIGQUIT to already running apps or passing in **-XX:DumpGCPerformanceOnShutdown** to **dalvikvm** when starting a command line program. When an app gets the ANR request signal (SIGQUIT) it dumps information related to its locks, thread stacks, and GC performance.

The way to get GC timing dumps is to use:
$ adb shell kill -S QUIT

This creates a **traces.txt** file in **/data/anr/**. This file contains some ANR dumps as well as GC timings. You can locate the GC timings by searching for: "Dumping cumulative Gc timings" These timings will show a few things that may be of interest. It will show the histogram info for each GC type's phases and pauses. The pauses are usually more important to look at. For example:

```
sticky concurrent mark sweep paused:    Sum: 5.491ms 99% C.I. 1.464ms-2.133ms Avg: 1.830ms Max: 2.133ms
```

This shows that the average pause was 1.83ms. This should be low enough to not cause missed frames in most applications and shouldn't be a concern.

Another area of interest is time to suspend. What time to suspend measures is how long it takes a thread to reach a suspend point after the GC requests that it suspends. This time is included in the GC pauses, so it is useful to determine if long pauses are caused by the GC being slow or the thread suspending slowly. Here is an example of what a normal time to suspend resembles on a Nexus 5:

```
suspend all histogram:  Sum: 1.513ms 99% C.I. 3us-546.560us Avg: 47.281us Max: 601us
```

There are also a few other areas of interest, such as total time spent, GC throughput, etc. Examples:

```
Total time spent in GC: 502.251ms
Mean GC size throughput: 92MB/s
Mean GC object throughput: 1.54702e+06 objects/s
```

Here is an example of how to dump the GC timings of an already running app:

```
$ adb shell kill -s QUIT <pid>
$ adb pull /data/anr/traces.txt
```

At this point the GC timings are inside of traces.txt. Here is example output from Google maps:

```
Start Dumping histograms for 34 iterations for sticky concurrent mark sweep
ScanGrayAllocSpaceObjects:      Sum: 196.174ms 99% C.I. 0.011ms-11.615ms Avg: 1.442ms Max: 14.091ms
FreeList:       Sum: 140.457ms 99% C.I. 6us-1676.749us Avg: 128.505us Max: 9886us
MarkRootsCheckpoint:    Sum: 110.687ms 99% C.I. 0.056ms-9.515ms Avg: 1.627ms Max: 10.280ms
SweepArray:     Sum: 78.727ms 99% C.I. 0.121ms-11.780ms Avg: 2.315ms Max: 12.744ms
ProcessMarkStack:       Sum: 77.825ms 99% C.I. 1.323us-9120us Avg: 576.481us Max: 10185us
(Paused)ScanGrayObjects:        Sum: 32.538ms 99% C.I. 286us-3235.500us Avg: 986us Max: 3434us
AllocSpaceClearCards:   Sum: 30.592ms 99% C.I. 10us-2249.999us Avg: 224.941us Max: 4765us
MarkConcurrentRoots:    Sum: 30.245ms 99% C.I. 3us-3017.999us Avg: 444.779us Max: 3774us
ReMarkRoots:    Sum: 13.144ms 99% C.I. 66us-712us Avg: 386.588us Max: 712us
ScanGrayImageSpaceObjects:      Sum: 13.075ms 99% C.I. 29us-2538.999us Avg: 192.279us Max: 3080us
MarkingPhase:   Sum: 9.743ms 99% C.I. 170us-518us Avg: 286.558us Max: 518us
SweepSystemWeaks:       Sum: 8.046ms 99% C.I. 28us-479us Avg: 236.647us Max: 479us
MarkNonThreadRoots:     Sum: 5.215ms 99% C.I. 31us-698.999us Avg: 76.691us Max: 703us
ImageModUnionClearCards:        Sum: 2.708ms 99% C.I. 26us-92us Avg: 39.823us Max: 92us
ScanGrayZygoteSpaceObjects:     Sum: 2.488ms 99% C.I. 19us-250.499us Avg: 37.696us Max: 295us
ResetStack:     Sum: 2.226ms 99% C.I. 24us-449us Avg: 65.470us Max: 452us
ZygoteModUnionClearCards:       Sum: 2.124ms 99% C.I. 18us-233.999us Avg: 32.181us Max: 291us
FinishPhase:    Sum: 1.881ms 99% C.I. 31us-431.999us Avg: 55.323us Max: 466us
RevokeAllThreadLocalAllocationStacks:   Sum: 1.749ms 99% C.I. 8us-349us Avg: 51.441us Max: 377us
EnqueueFinalizerReferences:     Sum: 1.513ms 99% C.I. 3us-201us Avg: 44.500us Max: 201us
```

```
ProcessReferences:      Sum: 438us 99% C.I. 3us-212us Avg: 12.882us Max: 212us
ProcessCards:   Sum: 381us 99% C.I. 4us-17us Avg: 5.602us Max: 17us
PreCleanCards:  Sum: 363us 99% C.I. 8us-17us Avg: 10.676us Max: 17us
ReclaimPhase:   Sum: 357us 99% C.I. 7us-91.500us Avg: 10.500us Max: 93us
(Paused)PausePhase:     Sum: 312us 99% C.I. 7us-15us Avg: 9.176us Max: 15us
SwapBitmaps:    Sum: 166us 99% C.I. 4us-8us Avg: 4.882us Max: 8us
(Paused)ScanGrayAllocSpaceObjects:      Sum: 126us 99% C.I. 14us-112us Avg: 63us Max: 112us
MarkRoots:      Sum: 121us 99% C.I. 2us-7us Avg: 3.558us Max: 7us
(Paused)ScanGrayImageSpaceObjects:      Sum: 68us 99% C.I. 68us-68us Avg: 68us Max: 68us
BindBitmaps:    Sum: 50us 99% C.I. 1us-3us Avg: 1.470us Max: 3us
UnBindBitmaps:  Sum: 49us 99% C.I. 1us-3us Avg: 1.441us Max: 3us
SwapStacks:     Sum: 47us 99% C.I. 1us-3us Avg: 1.382us Max: 3us
RecordFree:     Sum: 42us 99% C.I. 1us-3us Avg: 1.235us Max: 3us
ForwardSoftReferences:  Sum: 37us 99% C.I. 1us-2us Avg: 1.121us Max: 2us
InitializePhase:        Sum: 36us 99% C.I. 1us-2us Avg: 1.058us Max: 2us
FindDefaultSpaceBitmap: Sum: 32us 99% C.I. 250ns-1000ns Avg: 941ns Max: 1000ns
(Paused)ProcessMarkStack:       Sum: 5us 99% C.I. 250ns-3000ns Avg: 147ns Max: 3000ns
PreSweepingGcVerification:      Sum: 0 99% C.I. 0ns-0ns Avg: 0ns Max: 0ns
Done Dumping histograms
sticky concurrent mark sweep paused:    Sum: 63.268ms 99% C.I. 0.308ms-8.405ms
Avg: 1.860ms Max: 8.883ms
sticky concurrent mark sweep total time: 763.787ms mean time: 22.464ms
sticky concurrent mark sweep freed: 1072342 objects with total size 75MB
sticky concurrent mark sweep throughput: 1.40543e+06/s / 98MB/s
Total time spent in GC: 4.805s
Mean GC size throughput: 18MB/s
Mean GC object throughput: 330899 objects/s
Total number of allocations 2015049
Total bytes allocated 177MB
Free memory 4MB
Free memory until GC 4MB
Free memory until OOME 425MB
Total memory 90MB
Max memory 512MB
Zygote space size 4MB
Total mutator paused time: 229.566ms
Total time waiting for GC to complete: 187.655us
```

Tools for analyzing GC correctness problems

There are various things that can cause crashes inside of ART. Crashes that occur reading or writing to object fields may indicate heap corruption. If the GC crashes when it is running, it could also point to heap corruption. There are various things that can cause heap corruption, the most common cause is probably incorrect app code. Fortunately, there are tools to debug GC and heap-related crashes. These include the heap verification options specified above, valgrind, and CheckJNI.

CheckJNI

Another way to verify app behavior is to use CheckJNI. CheckJNI is a mode that adds additional JNI checks; these aren't enabled by default due to performance reasons. The checks will catch a few errors that could cause heap corruption such as using invalid/stale local and global references. Here is how to enable CheckJNI:

```
$ adb shell setprop dalvik.vm.checkjni true
```

Forcecopy mode is another part of CheckJNI that is very useful for detecting writes past the end of array regions. When enabled, forcecopy causes the array access JNI functions to always return copies with red zones. A *red zone* is a region at the end/start of the returned pointer that has a special value, which is verified when the array is released. If the values in the red zone don't match what is expected, this usually means a buffer overrun or underrun occurred. This would cause CheckJNI to abort. Here is how to enable forcecopy mode:

```
$ adb shell setprop dalvik.vm.jniopts forcecopy
```

One example of an error that CheckJNI should catch is writing past the end of an array obtained from **`GetPrimitiveArrayCritical`**. This operation would very likely corrupt the Java heap. If the write is within the CheckJNI red zone area, then CheckJNI would catch the issue when the corresponding **`ReleasePrimitiveArrayCritical`** is called. Otherwise, the write will end up corrupting some random object in the Java heap and possibly causing a future GC crash. If the corrupted memory is a reference field, then the GC may catch the error and print a "*Tried to mark not contained by any spaces*" error.

This error occurs when the GC attempts to mark an object for which it can't find a space. After this check fails, the GC traverses the roots and tries to see if the invalid object is a root. From here, there are two options: The object is a root or a non-root object.

Valgrind

The ART heap supports optional valgrind instrumentation, which provides a way to detect reads and writes to and from an invalid heap address. ART detects when the app is running under valgrind and inserts red zones before and after each object allocation. If there are any reads or writes to these red zones, valgrind prints an error. One example of when this could happen is if you read or write past the end of an array's elements while using direct array access through JNI. Since the AOT compilers use implicit null checks, it is recommended to use eng builds for running valgrind. Another thing to note is that valgrind is orders of magnitude slower than normal execution.

Here is an example use:

```
# build and install
$ mmm external/valgrind
$ adb remount && adb sync
# disable selinux
$ adb shell setenforce 0
$ adb shell setprop wrap.com.android.calculator2
"TMPDIR=/data/data/com.android.calculator2 logwrapper valgrind"
# push symbols
$ adb shell mkdir /data/local/symbols
$ adb push $OUT/symbols /data/local/symbols
$ adb logcat
```

Invalid root example

In the case where the object is actually an invalid root, it will print some useful information: **`art E 5955 5955 art/runtime /gc/collector/mark_sweep.cc:383] Tried to mark 0x2 not contained by any spaces`**

```
art E  5955  5955 art/runtime/gc/collector/mark_sweep.cc:384] Attempting see if
it's a bad root
art E  5955  5955 art/runtime/gc/collector/mark_sweep.cc:485] Found invalid
root: 0x2
art E  5955  5955 art/runtime/gc/collector/mark_sweep.cc:486]
Type=RootJavaFrame thread_id=1 location=Visiting method 'java.lang.Object
com.google.gwt.corp.collections.JavaReadableJsArray.get(int)' at dex PC 0x0002
(native PC 0xf19609d9) vreg=1
```

In this case, **`vreg 1`** inside of **`com.google.gwt.corp.collections.JavaReadableJsArray.get`** is supposed to contain a heap reference but actually contains an invalid pointer of address **`0x2`**. This is clearly an invalid root. The next step to debug this issue would be to use **`oatdump`** on the oat file and look at the method that has the invalid root. In this case, the error turned out to be a compiler bug in the x86 backend. Here is the changelist that fixed it: https://android-review.googlesource.com/#/c/133932/ (https://android-review.googlesource.com/#/c/133932/)

Corrupted object example

In the case where the object isn't a root, output similar to the following prints:

```
01-15 12:38:00.196  1217  1238 E art     : Attempting see if it's a bad root
01-15 12:38:00.196  1217  1238 F art     :
art/runtime/gc/collector/mark_sweep.cc:381] Can't mark invalid object
```

When heap corruption isn't an invalid root, it is unfortunately hard to debug. This error message indicates that there was at least one object in the heap that was pointing to the invalid object.
