Garbage Collection Workload for Android*
A New Way to Measure Android Garbage Collection Performance

As the mobile computing market evolves, Google Android* has become one of the most popular software stacks for smart mobile devices. Because Java* is the primary implementation language for Android applications, the Java Virtual Machine (JVM) is key to providing the best Android user experience.

The garbage collector (GC) component of JVM is one of its most important. It provides automatic memory management, and ensures that Java programmers cannot accidentally (or purposely) crash the JVM by incorrectly freeing memory, improving both programmer productivity and system security. GC keeps track of the objects currently referenced by the Java program, so it can reclaim the memory occupied by unreferenced objects, also known as "garbage." Accurate and efficient garbage collection is critical to reaching high performance and to the user experience of Android applications.

GC may interfere with user thread execution for garbage collection by introducing overhead and hence hurting performance and user experience. A typical example is that most GC algorithms require pausing all Java threads in an application at some point in order to guarantee that only garbage object memory is reclaimed. If the pause time is long, it can cause performance and user experience issues such as jank (unresponsive user interface or momentary sluggishness) and lack of responsiveness and smoothness. In addition, GC is usually triggered automatically by the JVM in the background, so programmers have little or no control over GC scheduling. Android developers must be aware of this hidden software component.

To achieve the best performance and user experience by optimizing GC, a workload that reflects GC performance is indispensable. However, in our experience most popular Android workloads (gaming, parsing, security, etc.) stress GC only intermittently. Intel developed Garbage Collection Workload for Android (GCW for Android) to analyze GC performance and its influence on Android performance and user experience.

GCW for Android stresses the memory management subsystem of the Android Runtime (ART). It is designed to be representative of the peak memory use behavior of Android applications, so optimizations based on GCW for Android analysis not only improve the workload score but also improve user experience. Further, GCW for Android provides options for you to adjust its behavior, making it flexible enough to mimic different kinds of application behavior.

GCW for Android Overview

Intel developed GCW for Android based on the analysis of real-world applications, including typical Android applications in categories such as business, communications, and entertainment. GCW for Android is an object allocation and GC intensive workload designed for GC performance evaluation and analysis.

The workload has two working modes. You can run it from the command line or control it using a GUI. GCW for Android is configurable. You can specify the workload size, number of allocation threads, object allocation size distribution, and object lifetime distribution to fit different situations. It provides flexibility to test GC with different allocation behaviors, so you can use GCW for Android to analyze GC performance for most usage situations.

GCW for Android incorporates several metrics. It reports the total execution time as the primary metric of GC and object allocation efficiency. The workload also reports memory usage information such as the Java heap footprint and total allocated object bytes based on the Android logging system.

How to Run GCW for Android

For the Android platform, GCW for Android is provided as a single package: GCW_Android.apk. After installing the apk, clicking the GCW_Android icon launches the workload and displays a UI that includes start and setting buttons.

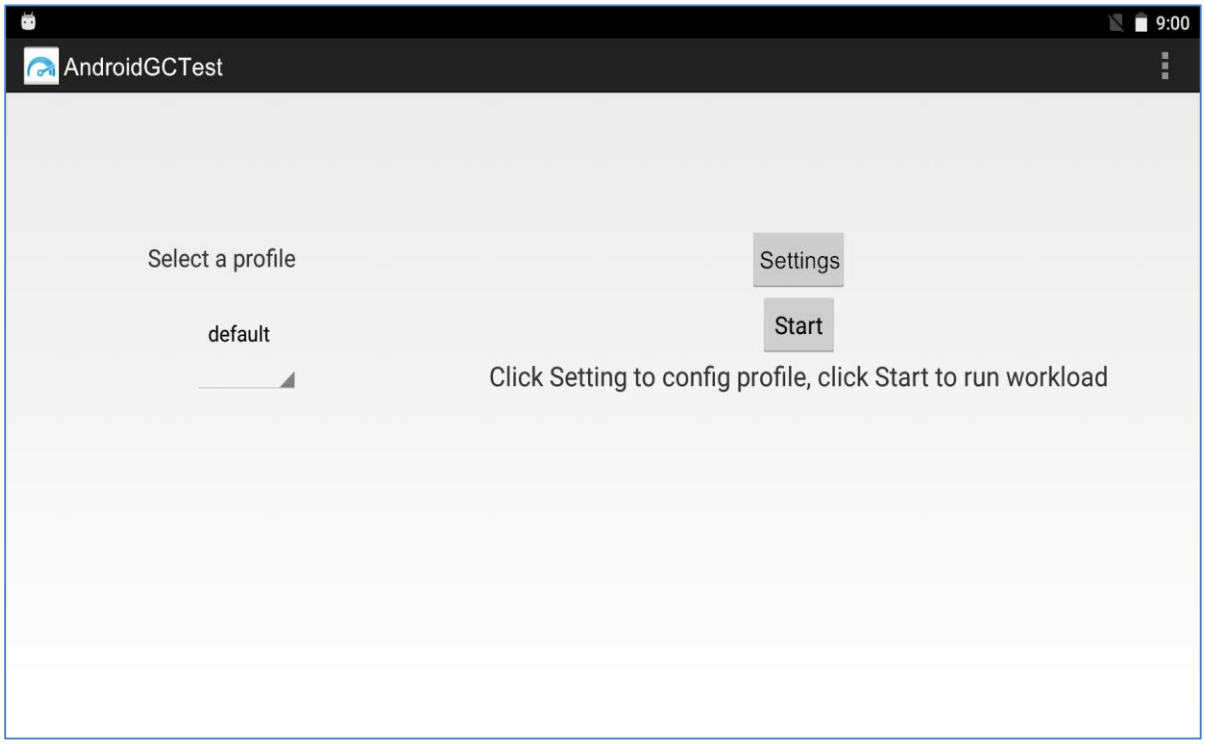
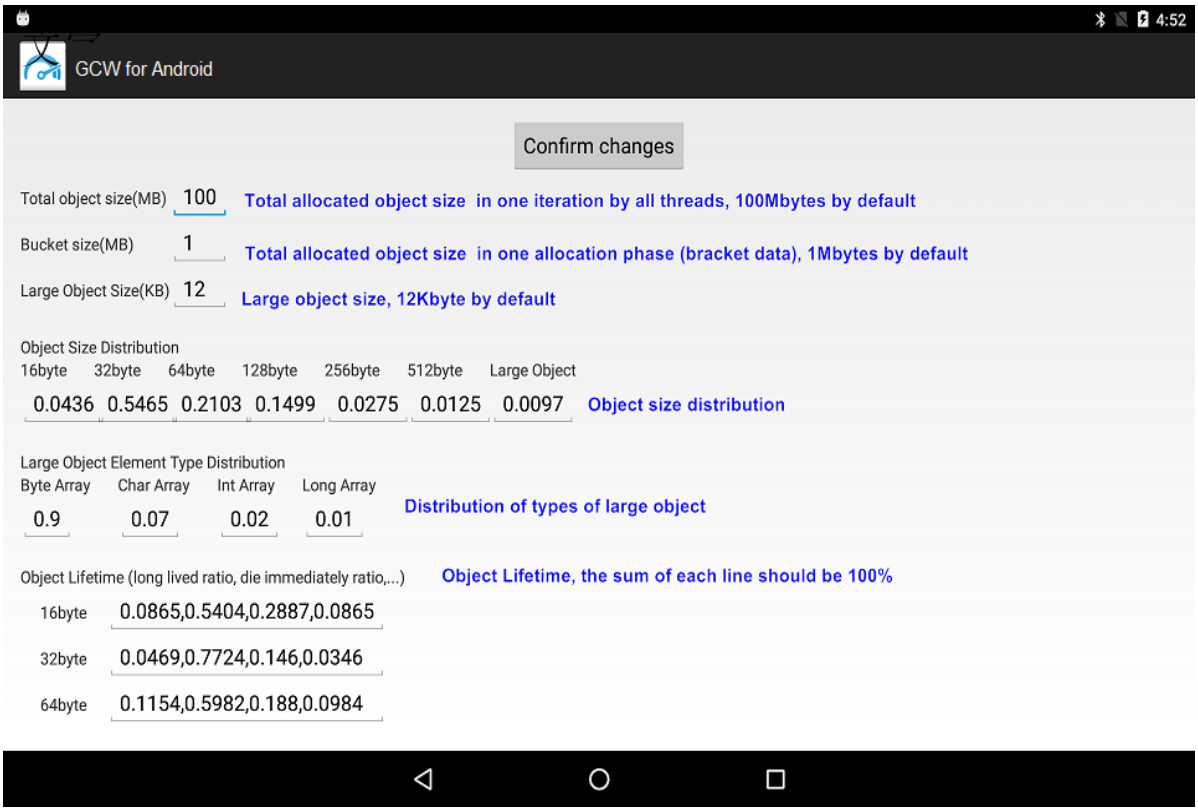
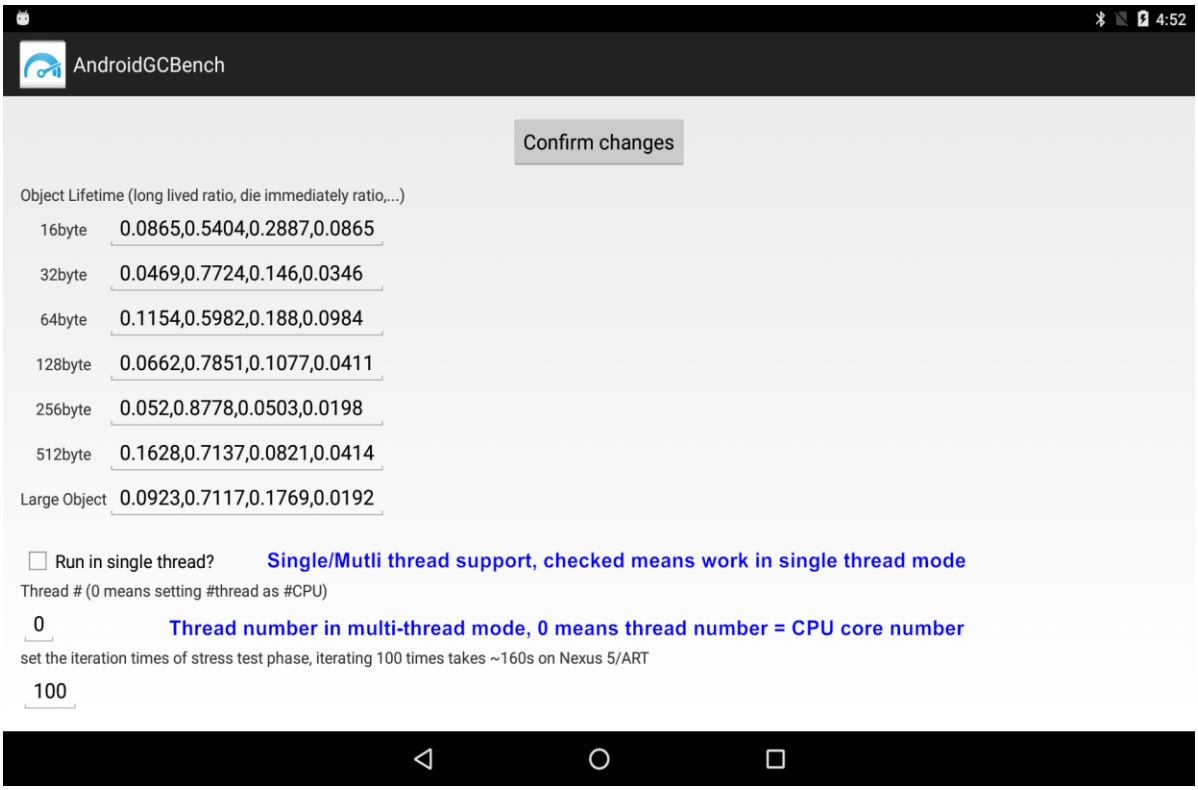


Figure 1. *GCW for Android Launch UI*

Clicking the “Start” button will run the workload using the default profile settings. The “default” settings option is used to reset the workload profile to the default. If you want to change the configuration, click the “Settings” button.



(a)



(b)

Figure 2. *Configuration UI. (a) is the top part (b) is the bottom part*

The first setting is a selectable list of profiles. For now, the “default” is the only option. A profile consists of the parameters used by the workload, and the default profile is derived from the characteristics of several real applications.

Total Object size: Allows you to define the total object size allocated in one iteration by all allocation threads. The default is 100MB, which means that when running in multi-thread mode with four threads, each thread will allocate 25MB’s worth of objects in one iteration in the stress test phase.

Bucket size: allows you to define the size of the binary trees that are built in a single allocation phase. The default is 1MB.

Large object size: allows you to define the large object size. The default is 12KB, which is the minimum size object that ART allocates in the large object space.

Object size distribution: allows you to define the size distribution of allocated objects. The total sum should be 100%.

Large object element type distribution: allows you to define the lifetime for each object. An object’s lifetime is defined in units of the size of the objects (1MB by default) allocated from when it is created to when it is made unreachable. The first item in the lifetime data is the long lived object percentage (the percentage of objects that live for the entire workload run), the second is the percentage of objects that die after the first period, the third is the percentage of objects that die after the second period (after allocating another 1MB worth of objects), and the K’th item is the percentage of objects that die after the K+1’st period (after allocating K MB’s worth of objects). Items are separated by commas and each line should have the same number of items.

By default, the workload runs in multi-thread mode. If you want to run in single-thread mode, check "Run in single thread?"

To understand how GCW for Android reflects the JVM’s memory management characteristics and is representative of real applications, let’s go deeper to see how GCW for Android is designed.

GCW for Android Design

GCW for Android is designed to mimic the JVM memory management behavior that real applications exhibit. Detailed analysis of different user scenarios on a large number of popular Android applications indicate that Java programs create various sized objects with varying lifetimes, so the workload does too. Also, the default relationship between object sizes and lifetimes (small objects tend to have short lifetimes) and the multi-threaded allocation behavior are similar to those of real applications.

Abstracted from the analysis data, the following characteristics were chosen as the primary design points for GCW for Android.

- **Object size distribution**

Figure 3 shows a histogram of object size distribution based on 17 popular Google Play* store applications. The X-axis is the percent of all objects of a given object size range and the Y-axis is the object size range buckets.

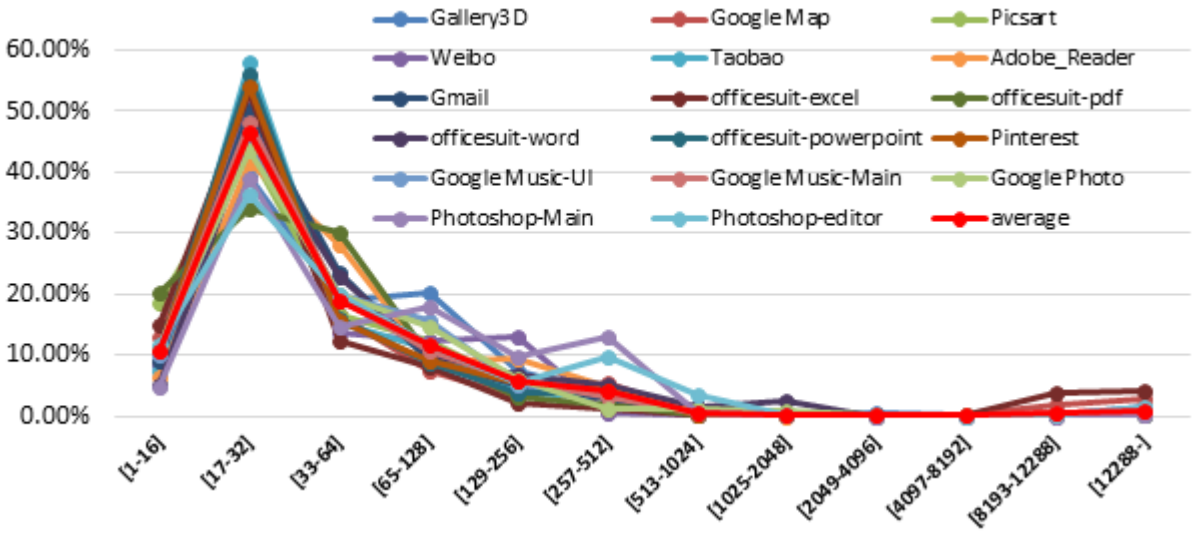


Figure 3. *Object size distribution of popular apps*

When running, around 80% of the objects allocated are small objects whose size is less or equal to 64 bytes. We observed the same behavior on ~50 more popular apps, so GCW for Android uses objects with many different object sizes. How GCW for Android models object size will be discussed in the next section “GCW for Android Workflow.”

- **Object lifetime**

Object lifetime is measured by how many garbage collections an object survives in the Java heap. Understanding object lifetime can help the JVM developer optimize GC performance by optimizing how GC works and tuning GC-related options.

In our investigations, the lines plotting object size against object lifetime show a loose relationship between the two, but lifetime does seem to be related to the object size. Here we choose Gallery3D (Figure 4) and Google Maps (Figure 5) as examples. The X-axis is the percentage of objects that die; the Y-axis is the number of GCs that survive. Each line represents an object size range.

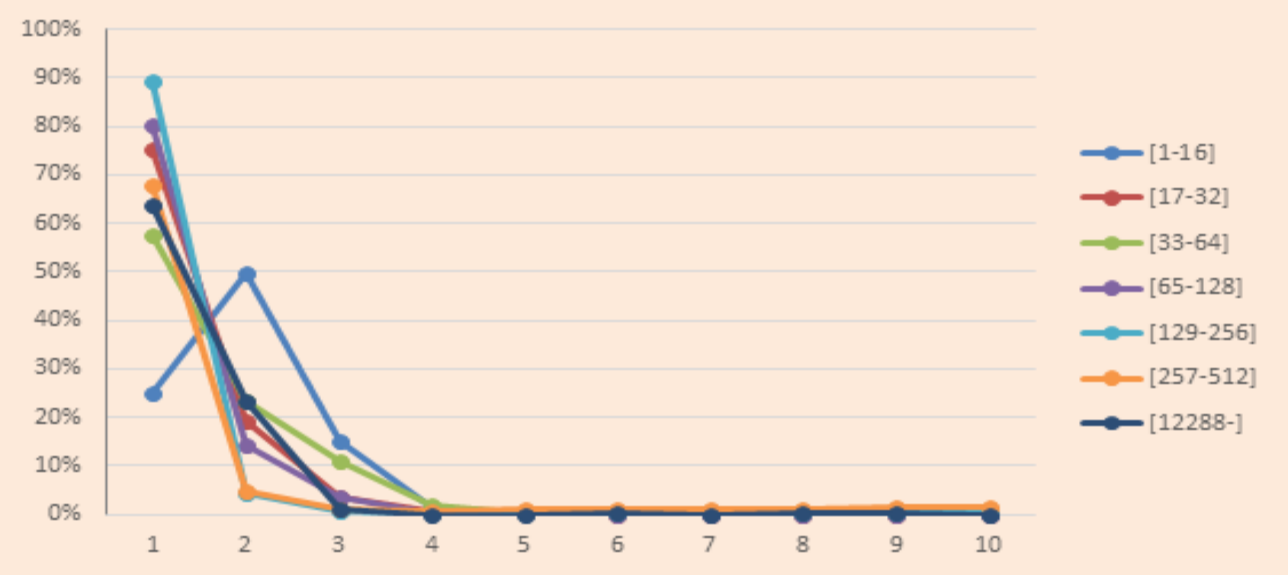


Figure 4. *Gallery3D Object Lifetimes*

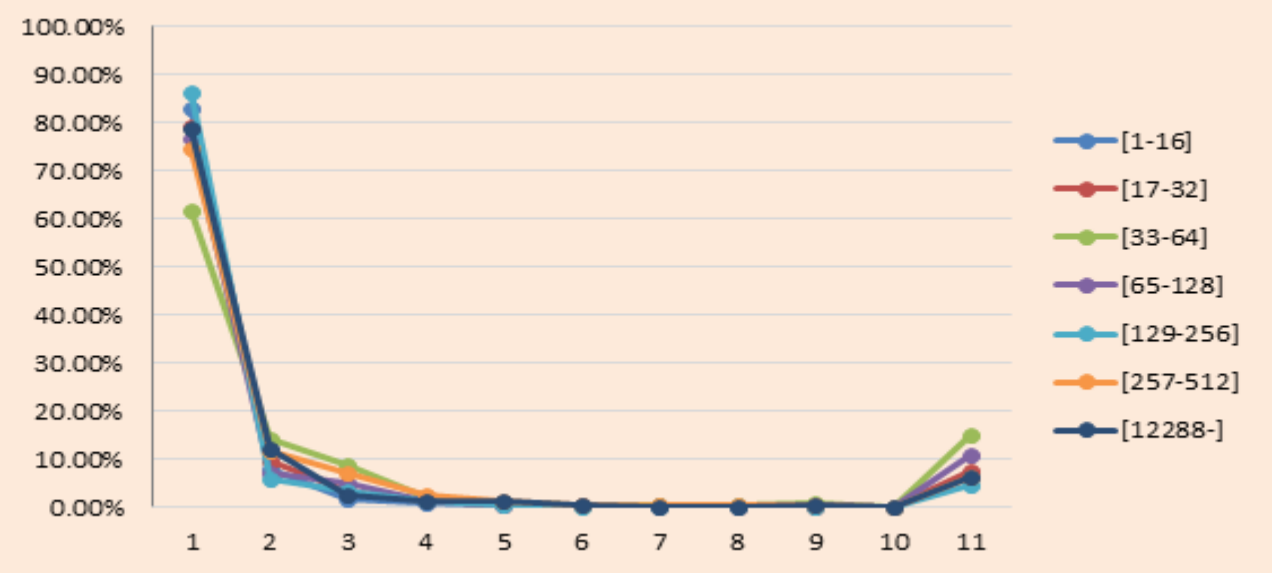


Figure 5. *Google Maps* mapping service Object Lifetimes*

Most objects die after one to three GCs, but the lines aren't quite congruent. For Google Maps, ~80% of the objects between 1-16 bytes die after the first GC, but only 60% of the objects between 33-64 bytes die after the first GC. Different-sized objects can have different lifetimes, so how well GCW for Android reflects object lifetime is an important design goal. How GCW for Android models object lifetime will be discussed in the next section "GCW for Android Workflow."

- **Multi-threading**

Our investigation shows that most Android Java applications are multi-threaded, though the threads do not typically communicate much with each other. Each Java application running on an Android device may have more than one thread allocating objects in the Java heap simultaneously. GCW for Android supports multi-threaded allocation in order to mimic this real application characteristic. How GCW for Android supports multi-threading will be explained in next section “GCW for Android Workflow.”

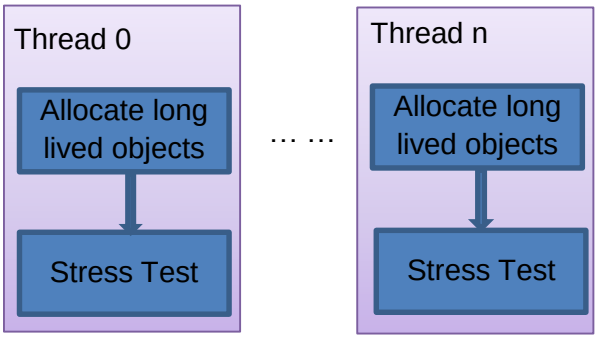
To summarize, the following workload characteristics are emulated in GCW for Android. Typical Android applications:

- Allocate varied size objects.
- Have similar allocated object size distributions.
- Have allocated objects with different lifetimes, and their lifetimes seem to be related to their size.
- Allocate objects in parallel in multiple threads.

Putting all these observations together, we designed the GCW for Android workflow to make it not only a JVM memory management workload, but also one that reflects actual usage scenarios.

GCW for Android Workflow

GCW for Android supports two threading modes: single- and multi-thread. While in multi-thread mode each thread is assigned a number, which is by default the logical CPU number. You can change it.



(a)

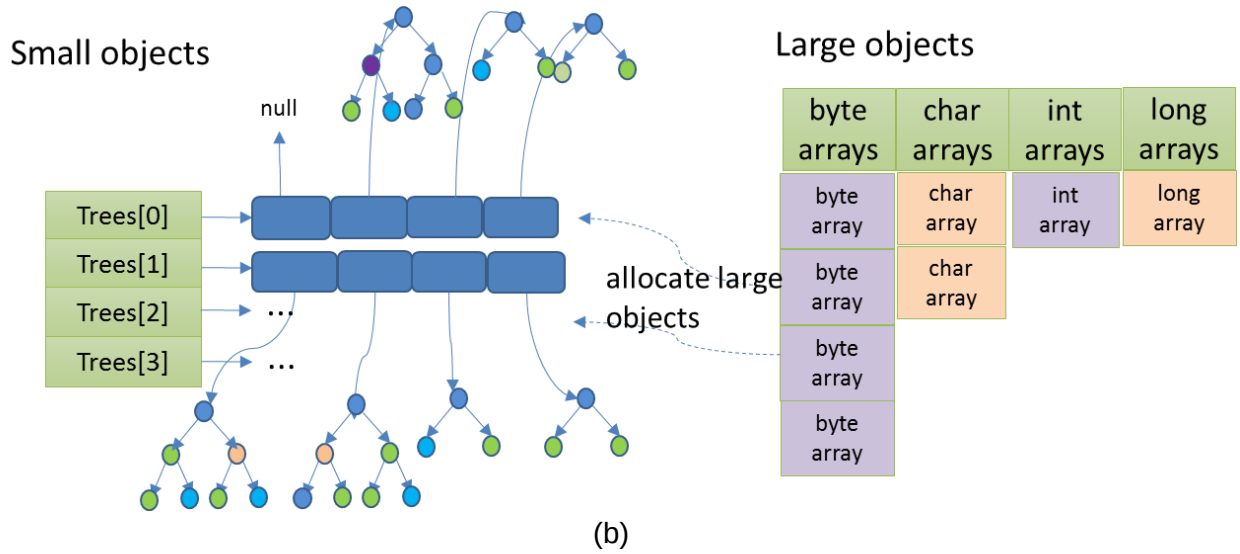


Figure 6. *GCW for Android Workflow*

Internally, GCW for Android builds several binary trees in order to manage object sizes and lifetimes. It builds and deletes objects by inserting and deleting nodes in the trees, as shown in Figure 6(b).

Figures 6(a) and (b) show how GCW for Android works. It first launches a certain number of threads, and then each thread follows the same logic: allocate long lived objects and then stress test. The stress test is the most time consuming part of the workload and is a big loop that iterates a configurable number of times, by default 100. In each iteration, GCW for Android allocates a configurable number of bytes (by default 100MB), which by default includes 6 kinds of small objects (16, 24, 40, 96, 192, and 336 byte objects) plus large objects (12K bytes, configurable). Small objects are created as nodes of binary trees; large objects are created as byte, char, int, or long arrays. In each iteration of GCW for Android, the workload:

- Builds binary trees.
- Deletes some nodes from built trees according to the lifetimes of small objects.
- Builds large object arrays.
- Deletes some arrays according to lifetimes of large objects.

Now let's take a closer look at the internal design of GCW for Android to understand how it achieves the aforementioned characteristics of emulating memory system use.

Object Size Distribution

To simulate common object size distribution patterns, GCW for Android internally defines seven object size buckets:

- 16 byte → [1-16B]

- 24 byte → [17-32B]
- 40 byte → [33-64B]
- 96 byte → [65-128B]
- 192 byte → [129-256B]
- 336 byte → [257-512B]
- Large object (default is 12KB)

To conveniently abstract object references, GCW for Android allocates small sized objects as nodes in a binary tree, and large objects are created as arrays. There are four types of large object arrays: byte, char, int, and long.

Now let's see how GCW for Android manages object lifetimes to mimic real applications.

Object Lifetime Control

GCW for Android internally uses binary trees to control object lifetime. Every binary tree has a predefined lifetime. GCW for Android controls tree lifetimes and therefore object lifetimes.

For example, suppose you want your object lifetime model to have three stages where 50% of objects die in period K, 25% of objects die after that in period K+1, and the remaining 25% live through period K+2.

At the beginning of period K, GCW for Android builds three trees simultaneously, one tree for each lifetime stage. In period K, one tree holding 50% of the objects is made unreachable by assigning null to the root object, which makes the whole tree unreachable from the GC point of view. So 50% of the objects are collected by GC after period K. Then in period K+1, the second tree holding 25% of objects is rendered unreachable by setting the root to null, thus causing 25% of the objects to be reclaimed after period K+1 and leaving 25% of the objects alive through period K+2 (see Figure 7).

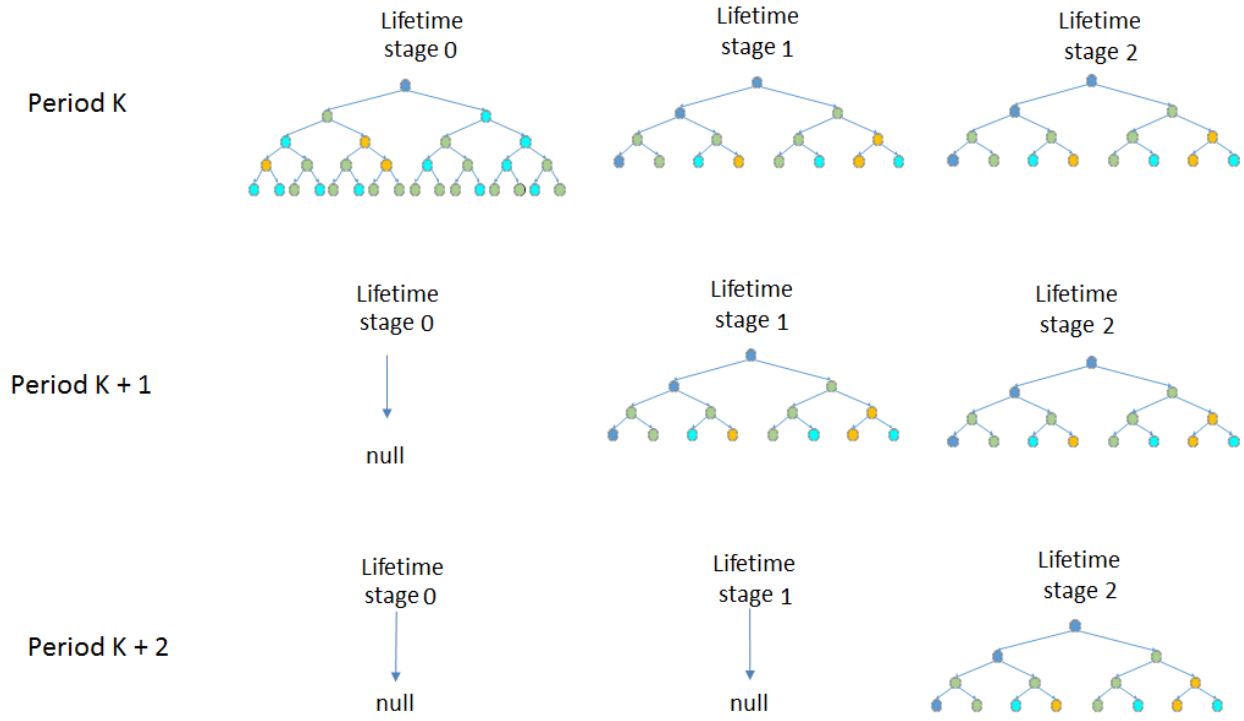


Figure 7. *Object lifetime control*

In different use scenarios, object lifetime is not deterministic, so GCW for Android also does not make object lifetimes deterministic. To emulate real applications, GCW for Android generates a random number for every thread, each with a different seed, in order to decide object size and the lifetime stage of a node. Here is an example.

object size	lifetime stage
24	1
96	0
16	0
24	0
24	0
16	0
24	1
24	1
24	0
40	0
16	0
24	0

.....

In this example three 24-byte objects will die after stage one, and the other objects will die after stage 0.

To recap, GCW for Android makes it easy to reflect the object allocation and GC behavior of real applications. Using GCW for Android can help you identify many opportunities in the ART to enhance performance and user experience.

Opportunities Discovered Using GCW for Android

During our performance investigation we discovered that object allocation is typically the hottest part of an application. Further investigation showed that allocation can be made faster by inlining the RosAlloc allocation fast path, which means the call to the allocation function is eliminated. That resulted in ~7% improvement on GCW for Android. The implementation has been merged into the Android Open Source Project (AOSP).

Additionally, we found that GC marking time can be reduced by eliminating unnecessary card table processing between immune spaces. The card table is a mechanism that records cross-space object references, which guarantee the correctness of GC when it collects only a subset of the Java heap (the subsets are called “spaces”) instead of the whole heap. Analysis also revealed that GC pause time can be reduced by clearing dirty cards for allocation spaces during partial and full GCs. Minimizing pause time is critical for end user apps because reducing pause time can reduce jank and improve smoothness.

GCW for Android also helped us find GC optimization opportunities such as parallel marking and parallel sweeping. Intel has contributed several of these optimizations to AOSP and the rest have been added to the Intel ART binaries. Altogether, these optimizations have resulted in ~20% improvement on GCW for Android. All have helped improve Intel product performance and user experience.

Open Source GCW for Android

We have submitted GCW for Android for upstreaming into AOSP to make it available to the entire Android development community. Our goal has been to make GCW for Android the most realistic Android Java memory system workload and use it to drive GC optimization on all platforms to improve Android performance and user experience.

Downloads

The code can be downloaded from Google at

https://android-review.googlesource.com/#/c/167279/

Conclusion

GCW for Android is a JVM memory management workload that is designed to emulate how real applications use Java memory management. It is intended to help both JVM and application developers optimize memory management and user experience on Android. Intel is using it to improve product performance and user experience by identifying optimization opportunities in ART. We hope it becomes an important indicator of performance and user experience on Android.

References

Java Virtual Machine (JVM): https://en.wikipedia.org/wiki/Java_virtual_machine
Garbage Collection (GC): https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)
Android Runtime (ART): https://source.android.com/devices/tech/dalvik/
Inside the Java Virtual Machine: https://www.artima.com/insidejvm/ed2/index.html

Acknowledgements (alphabetical)

Jean Christophe Beyler, Dong Yuan Chen, Haitao Feng, Jean-Philippe Halimi, Paul Hohensee, Aleksey Ignatenko, Rahul Kandu, Lei Li, Desikan Saravanan, Kumar Shiv, and Sushma Kyasaralli Thimmappa.

About the Authors

Li Wang is a software engineer in the Intel Software and Solutions Group (SSG), Systems Technologies & Optimizations (STO), Client Software Optimization (CSO). She focuses on Android workload development and memory management optimization in the Android Java runtime.

Lin Zang is a software engineer in the Intel Software and Solutions Group (SSG), Systems Technologies & Optimizations (STO), Client Software Optimization (CSO). He focuses on memory management optimization and functional stability in the Android Java runtime.

Notice

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.