This document corresponds to the version of the Marvin paper that was published as tech report UW-CSE-2019-04-01.

This document describes the code (apps, scripts, and Marvin versions) used to generate each of the graphs in the Marvin paper. Where possible, I have recorded the versions of the marvin-code and/or marvin-art repositories that were used when running each experiment. I expect each experiment to behave identically when using the latest versions of each repo.

For all graphs with commercial Android apps (figures 1, 2, and 4), I used an APK of Google Maps obtained by installing it from Google Play Store on 2/9/2018 and pulling the APK from my device with ADB. I similarly used an APK of the Amazon App Store that was installed and pulled on 10/5/2018. All other apps were installed from the Amazon App Store.

For experiments run on stock Android, I used AOSP compiled from source (tag android-7.1.1_r57).

Figure 1 (graph of heap percentage occupied by objects of a given size or smaller)
I collected the data by uncommenting the code in the marvin-art repo's file runtime/marvin_instrumentation.cc that prints out the objectSizeTotalMemoryCounts histogram. I converted that data to CDFs using scripts/histogram-to-cdf.pl, and I graphed it using scripts/graph-object-size-heap-frac-graph.R.

marvin-art commit 7fdb252

Figure 2 (Java heap size vs minimum Java read working set size)
Java heap sizes were measured on stock Android using the Android Studio Profiler. Working set data was measured on Marvin and processed using scripts/process-working-set-data.pl. I graphed the data using scripts/graph-min-working-set.R.

I did not record the marvin-art commit used, but I know that it was commit cf214cc or later.

Figure 3 (Android memory allocation speed)
I ran malloc-timer/malloc-timer.c on the device to allocate memory, and I ran memory-measurement/measure-memory.c on the device to measure malloc-timer's memory footprint. For the run with memory pressure, I enabled an 18GB Linux swapfile, started 8 clones of apps/MicroBenchmark modified to create 500 1MB arrays and disable the worker thread, dropped the device's disk cache (by running 'sync && echo 3 > /proc/sys/vm/drop_caches' in an ADB root shell), and then ran malloc-timer. I processed the data using scripts/process-memory-data.pl and graphed it using scripts/graph-android-allocation-data.R.

marvin-code commit 6cd0398

Figure 4 (App startup time vs disk read time)

I measured app startup times by recording the device's screen with a webcam and measuring the time difference between the frame in which the app startup animation began and the frame in which all in-app loading animations finished. Disk read times were extrapolated using app memory image sizes measured with the Android Studio Profiler and a disk read throughput measured on the device with ssd-benchmark/benchmark.c.

Figure 7 (Marvin memory reclamation)
I measured memory usage of apps/MicroBenchmark as it moved to the background, measuring memory footprints using memory-measurement/measure-memory.c. MicroBenchmark was changed to have largeHeap=true and have 500 1MB arrays. I processed the data using scripts/process-memory-data.pl and graphed it using scripts/graph-memory-data.R.

An empty Android app running on this version of Marvin had the following /proc/[pid]/statm reading, which I used to obtain the baseline for the graph:
591860 20856 11836 4 0 38214 0

marvin-art commit 3f0f082
marvin-code commit 590cd05

Figure 8 (Active apps over time)
Apps were clones of apps/MemoryWaster's AllocatorActivity (4KB runs) or its MixAllocatorActivity (4KB/1MB mix runs). I created and ran the clones with scripts/manage-clones.pl. I processed the resulting log files with scripts/parse-allocatoractivity-log.pl and scripts/process-parsed-allocatoractivity-logs.pl, and I graphed the data with scripts/graph-live-apps.R. I changed the $ACTIVITY_NAME variable in scripts/parse-allocatoractivity-log.pl when switching between the AllocatorActivity and MixAllocatorActivity runs.

In order to allow Marvin and Android with a swap file to run more than 17 apps concurrently, I increased the value of the constant MAX_CACHED_APPS in the file frameworks/base/services/core/java/com/android/server/am/ProcessList.java inside the AOSP source tree. This modification was used when compiling both stock Android and Marvin.

marvin-art commit ba719ff (4KB runs) and 3f0f082 (4KB/1MB mix runs)
marvin-code commit 41fa1db (4KB runs) and ca9d370 (4KB/1MB mix runs)

Figure 9 (PCMark for Android results)
I ran Marvin with commercial app compat mode enabled (by setting the corresponding macro to true in marvin-art's file runtime/marvin_swap.cc). For each system, I ran the PCMark for Android Work 2.0 benchmark five times, and reported the average and standard deviation for each benchmark. I graphed the data using scripts/graph-pcmark-data.R.

marvin-art commit 3f0f082

Figure 10 (Overhead with different proportions of OAI)
I ran apps/MicroBenchmarkTwo with different values of the INT_OP_LOOPS_PER_ITER and OBJ_OP_LOOPS_PER_ITER variables to get different DEX instruction mixes. I used dexdump to disassemble the DEX bytecode of the app, and I manually counted instructions inside of the timing loop to determine a formula converting values of the constants above into the fraction of DEX instructions with OAI. I used the Release build of the app so that its DEX bytecode was easily accessible. For each configuration, I let the timing loop run at least 45 times, threw away the first 5 iterations, and used the next 40 iterations. To compute the mean and standard deviations of Marvin's overhead, I paired the ith iteration of the timing loop for the Marvin run of a given configuration with the ith iteration for the corresponding Android run, divided the two values to get Marvin's overhead for the ith iteration, and then computed the mean and standard deviation of those overhead values. I did that computation using scripts/process-compiler-overhead-synthetic-data-paired.pl and graphed the data using scripts/graph-compiler-overhead-synthetic-variance.R.

marvin-art commit 3f0f082
marvin-code commit 078af71

Figures 11 (Speed of heap walk with different fractions of reclaimed objects)
I used apps/HeapWalker for this experiment. I disabled Marvin's preemptive swapping by setting the corresponding macro to false in marvin-art's file runtime/marvin_swap.cc. For each run, I let the app run in the foreground until all of its non-working set objects had been written to disk, moved the app to the background and waited until it had finished creating stubs, dropped the device's disk cache, and then moved the app to the foreground. Reported values are the means and standard deviations of five runs for each configuration.

marvin-art commit 3f0f082
marvin-code commit fff376c