The Java* Application Component Workload for Android*: Real Java* Application Use Cases for Android*

Introduction

The Android system has roughly a billion users, with almost a million applications from the Play Store downloaded and used daily. Android users equate their User Experience (UX) with their application (app) experience. Measuring UX via running Play Store apps is subjective, non-repeatable and difficult to analyze. Further, most magazine benchmarks don't stress critical paths representing real application behavior; optimizing for these benchmarks neither improves UX nor delights app users. Many Android benchmarks that have been used since Cupcake (Android 1.0) have become obsolete due to improvements in Android compilers and runtimes.

This paper discusses the Intel-developed Java* Application Component Workload (ACW) for Android*. ACW was developed to bridge the gap between magazine benchmarks and more sophisticated workloads that model real app behavior, as well as to provide a guide to robust optimizations that perceivably improve UX. The workload has been analyzed to help app developers write optimal Java code for Android (for more information, see https://software.intel.com/en-us/articles/how-to-optimize-java-code-in-android-marshmallow). OEMs, customers and system engineers can use ACW to compare Android software and System-on-a-Chip (SoC) capabilities

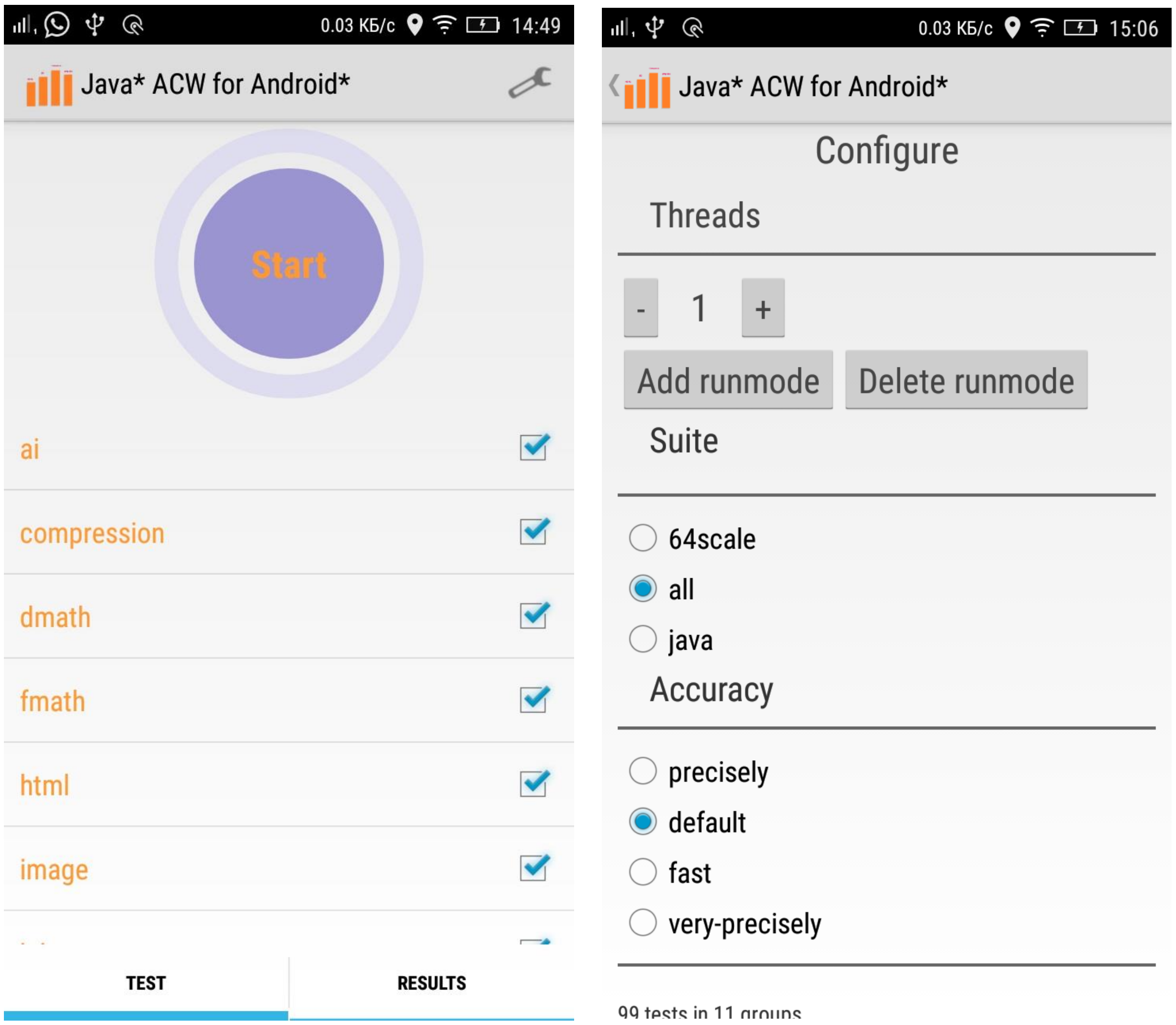
Android Component Workload (ACW) Overview

The workload consists of a set of computational kernels from often-used applications. Kernels are groups of tests in the areas of gaming, artificial intelligence, security, parsing HTML, PDF document parsing and encryption, image processing and compression/decompression. ACW stresses the Android Java Runtime (ART) compiler and runtime, measuring the impact of compiled code and its runtime overhead while executing the application program. VM engineers and performance analysts can use the workload to explore ways to improve ART code generation, object allocation and runtime optimization, as well as suggest ways to improve the micro-architecture of upcoming SoCs.

ACW includes a set of tests designed to measure the difference between 32- and 64-bit code, but the primary focus is on UX. ACW can be run from both a Graphical User Interface (GUI) and the command line (using adb shell). The workload can mix and match which kernels are run and measured. The use cases are of fixed duration and they report a throughput-based score. The final score (operations/second) is the geometric mean of the throughput scores of each kernel.

How to run ACW in the Android mobile environment (user mode)

On the Android platform, ACW is provided as an Android application package (apk). After installation, clicking on the JACWfA icon (see icon in Figure 1) launches the workload and displays a UI with the Start option at the center of the screen. Additional navigation is available at the bottom of the UI using two tabs: TEST and RESULTS. By default, the TEST tab will be displayed. Click the Start button to run the workload using the default profile settings. By default, all tests (tests are the kernels described above) will be selected, but the user may deselect any particular test or tests. In the top right corner, there is a Settings icon which allows users to configure Threads, Suite and Accuracy for each run.



ai ✓
compression ✓
dmath ✓
fmath ✓
html ✓
image ✓

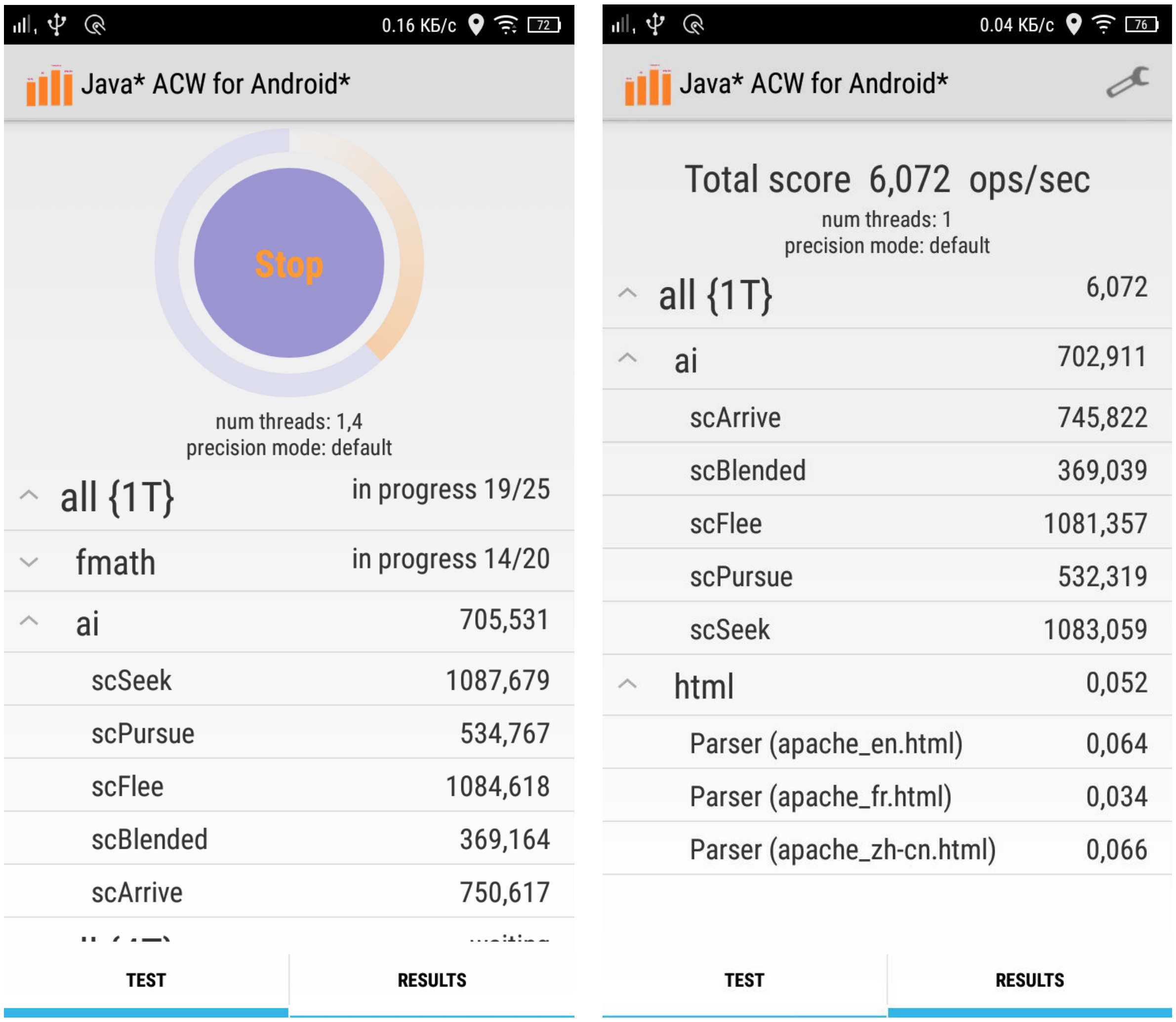
Threads
- 1 +

Suite
64scale
[✓ all]
java

Accuracy
precisely
[✓ default]
fast
very-precisely

99 tests in 11 groups

Figure 1. ACW UI *(from left to right: Main / Configure activities)*


0.16 КБ/с

Java* ACW for Android*

num threads: 1,4
precision mode: default

all {1T}	in progress 19/25
fmath	in progress 14/20
ai	705,531
scSeek	1087,679
scPursue	534,767
scFlee	1084,618
scBlended	369,164
scArrive	750,617

0.04 КБ/с

Java* ACW for Android*

Total score 6,072 ops/sec
num threads: 1
precision mode: default

all {1T}	6,072
ai	702,911
scArrive	745,822
scBlended	369,039
scFlee	1081,357
scPursue	532,319
scSeek	1083,059
html	0,052
Parser (apache_en.html)	0,064
Parser (apache_fr.html)	0,034
Parser (apache_zh-cn.html)	0,066

Figure 2. ACW UI *(from left to right: Progress / Results activities)*

The workload can also be run as an Android application from the command line via adb. For example:

```
adb shell am start -S –n com.intel.mttest.android/com.intel.mttest.android.StarterActivity -e
autostart true -e -s masterset:all
```

The Android version of the workload is designed to mimic the characteristics exhibited by real applications. Most Java applications in Android are multi-threaded, so the workload has support for as many threads as there are available cores on the device, though there is no direct communication between these threads.

ACW has three pre-configured test set options: Java, 64scale and All. Java mode runs only tests which closely mimic real application behavior. 64scale mode runs computational tests for the purpose of comparing 32- and 64-bit execution. All mode runs every test in the workload. Additionally, one can select how long tests should run via the Accuracy modes. They recognize that the longer a test runs, the more accurate will be the result. The four predefined Accuracy modes are: very-precisely (longest run), precisely, default and fast (shortest run). The default Accuracy is a balance between test result stability and time to run, and should take 20-40 minutes to complete on the default test set configuration. To modify accuracy settings the corresponding XML configuration files should be updated. When the configuration is complete, the user can return to the home UI by clicking on the icon on the top left corner or by using the back option at the bottom of the screen.

Click the START button to run the workload (Figure 3). The progress wheel (as on Figure 2) shows the run status.

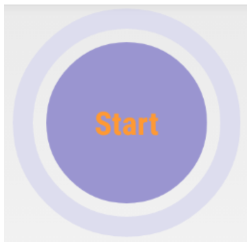


Figure 3. ACW progress wheel

The UI displays which tests have completed, the one currently in progress and which are yet to run. The final score is displayed at the top of the RESULTS tab. Additionally, individual test and subtest scores are displayed on the screen and are available from logcat messages. Note that ACW can be run in a Developer mode which allows even easier customization and debuggability. More information on Developer mode can found on the web at How to run ACW in Developer mode.

ACW Tests (Kernels)

ACW includes over 80 tests grouped into kernels (see the table below) associated with different application areas. Every kernel includes a number of tests that implement realistic Android application scenarios using standard Java libraries. Some tests, such as MATH and SORT, implement well-known algorithms.

Kernel	**Library**	**Description**
Artificial Intelligence (AI)	libGdx AI	Artificial intelligence

Compression	XZ, Apache Commons	Compression
Dmath		Decimal integer math algorithms
Fmath		Floating point math algorithms
Html	jsoup	Html parser
Image	BoofCV	Image processing
Jni		JNI stress
Lphysics	jBullet	Physics engine
Pdf	Pdfbox	Pdf parsing and encryption
Sort		Sort algorithms
Xphysics	jBox2D	Physics engine

ACW produces scores for individual kernels and their subtests and an overall performance score. The overall score is measured in operations per second (ops/sec). Here are the formulas to calculate scores:

Test Total = sum of executed Test iterations. *
Single Test score = Test Total / Time spent for execution.
Kernel score = GEOMEAN of the component tests scores.
Overall score = GEOMEAN of all kernel scores.
** Defined by test implementation*

The Design of ACW for Android

ACW uses the open source MTTest modular framework, which was specially developed for this workload. It can be configured to run under both ART and the Java Virtual Machine (JVM) on Android as well as Linux- and Windows-based PC environments using both 32- and 64-bit operating systems.

MTTest has 4 major modules:

- Configuration (configures workload)
- Runner (tests run functionality)
- Summary (collects test run results)
- Reporter (reports results in a specific way)

The framework is designed to be easily extended for new performance testing cases. Runner, Summary and Reporter modules can be easily updated to support new functionality, such as reporting results in XML file format.

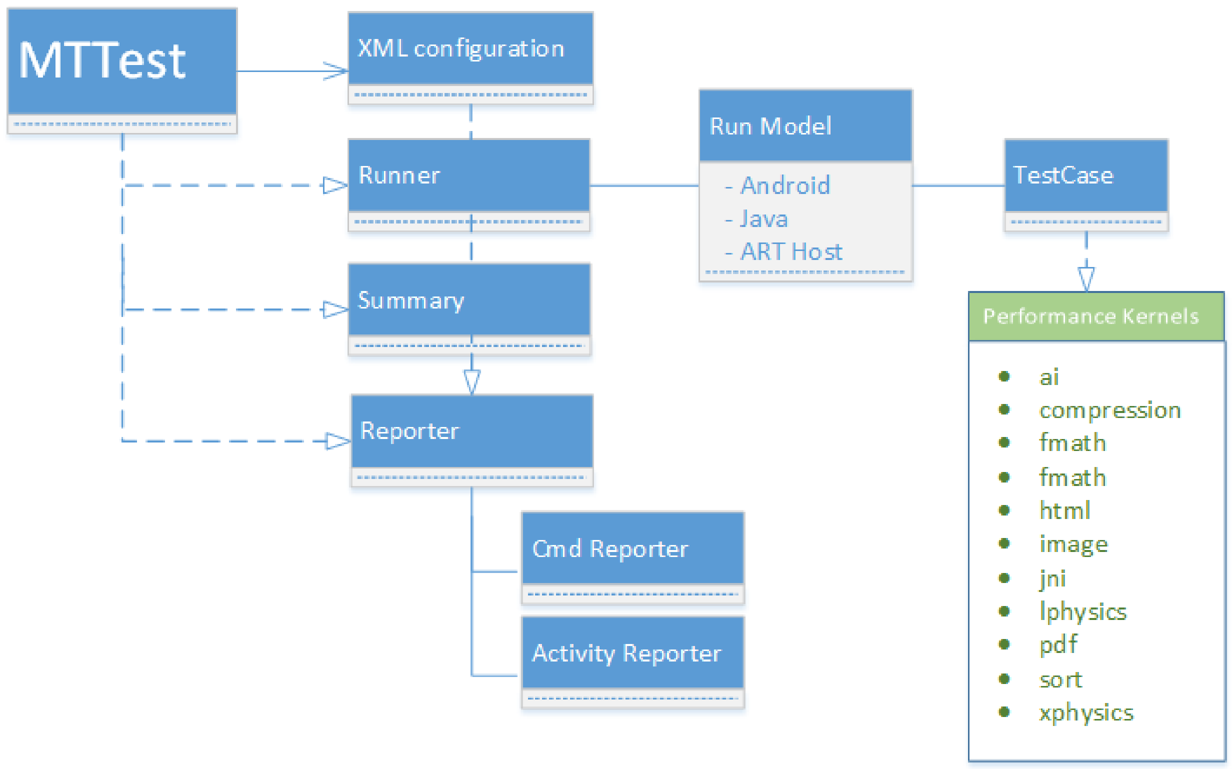


Figure 4. ACW architecture diagram

Standard workflow shown in the architecture diagram (Figure 4) is:

- Read the workload configuration and test set from XML configuration files
- Run test cases (run mode is defined by Run Model)
    - Perform test specific initialization
    - Run test (Three phases)
        - Ramp-Up
        - Measurement
        - Ramp-Down
- Summary module collects results

- Deliver results with a specific Reporter (Activity Reporter) for Android application

Test Ramp-Up is used to allow JVMs with Just-In-Time (JIT) compilers to compile hot code so that we can measure pure JVM performance. That is, we wait long enough to trigger JIT compiles so the main test is run using compiled code, not the interpreter. Ramp-up and ramp-down phases are also needed for multi-threaded runs in order to ensure that all test threads are running while some of them are starting to ramp-up or others are finishing measurement. The Measurement phase is used to compute results based on test duration and number of operations executed.

In all tests, iteration() executes in a loop until the time spent reaches the configured value of stageDuration, which limits test run time (iteration time):

```
long startTime = System.nanoTime();
do {
      count++;
      score += test.iteration();
} while(System.nanoTime() < startTime + stageDuration);
elapsedTime = System.nanoTime() - startTime;
summary.collect(score, elapsedTime , count);
```

Ramp-up/down times are defined by two types of configuration files. The first is the Test configuration file (<*ACW_DIR*>/assets/testsets/masterset.xml) which includes a list of XML files which describe the tests to be run and their parameters. In the example below, pdf.xml describes the Pdf kernel, subtest names, number of repetitions and input file names.

```
<?xml version="1.0"?>
<mttest version="0.1">
<conf name="timeUnits" value="second" />
<workload name="com.intel.JACW.pdf.Encryption">
    <option name="repeats" value="1" />
    <option name="goldenFileName" value="apache.pdf" />
</workload>
<workload name="com.intel.JACW.pdf.Parser">
    <option name="repeats" value="1" />
    <option name="goldenFileName" value="apache.pdf" />
</workload>
</mttest>
```

With a default test set configuration, the workload runs certain tests several times and compares the result against different type/size golden files (<*ACW dir*>/assets/goldens). This is done to emulate system load variation as on real end user systems. As a result, the default number of test runs is more than the number of performance tests.

The second type of XML file is a setup configuration file (<*ACW dir*>/assets/configs) that limits test run times. There are four predefined XML configuration files: short, medium, long and very_long. By default, medium.xml is used to limit workload run time up to six seconds, ramp-up to two seconds and ramp-down to one second:

```
<?xml version="1.0"?>
<mttest>
    <name value="default" />
    <conf name="rampUp" value="2000" />
    <conf name="duration" value="6000" />
    <conf name="rampDown" value="1000" />
    <conf name="isValidating" value="false"/>
</mttest>
```

ACW for Android Performance Overview

ACW runs multiple threads (up to the number of CPU cores) as part of its default settings. Real Android applications in Java are often multi-threaded, although interaction between multiple Java threads is a new area of focus among Android performance analysts and is not present in ACW. From a performance investigation standpoint, we have focused on the single-threaded case, since app threads typically do not interact except during synchronization within the ART framework libraries.

Performance analysis is motivated towards guiding VM engineers to identify optimizations for Android UX and understand the SoC limitations that impact ART performance. ACW's ART use-cases that mimic real Java application behavior (HTML, Pdf, Lphysics, Ai, Image and Compression) spend most of their execution time in ART compiler generated code for the application, ART framework system library code, and the ART runtime. ART runtime overhead is commonly associated with object allocation, array bound check elimination, class hierarchy checks and synchronization (locks). These applications use typical ART framework library (libcore) routines from java.lang.String, StringBuffer, Character, java.util.ArrayList, Arrays, and Random. A small amount of time is spent in native String allocation.

While delving into SoC characterization, we have seen Instruction Translation Lookaside Buffer (ITLB) cycle miss costs of 8-14% (lost CPU time), and 3-5% Data Translation Lookaside Buffer (DTLB) cycle miss costs. Performance is often restricted by instruction cache size limitations on devices with smaller instruction caches and is processor front-end (instruction parsing and functional unit distribution) bound.

ACW Optimization Opportunities in ART

ACW opens the door for Profile Guided Optimizations (PGO) done by most Java optimizing compilers e.g. de-virtualization, call site specialization, direct call conversion and method inlining in ART. We have also identified opportunities for loop transformations such as loop fusion and unrolling, and null check elimination and array bound check elimination within loops. Further opportunities include use of intrinsics for a few methods from java.lang.String and java.util.math, and native inlining for methods that call into ART's native String implementation.

Open Source ACW for Android

ACW has been open sourced at https://github.com/android-workloads/JACWfA/ as part of Intel's contribution to improving the way User Experience performance is measured on Android. After optimizing away several synthetic benchmarks such as CF-Bench and Quadrant, Intel's team took a step forward by using Icy Rocks* (see the paper for details https://software.intel.com/en-us/android/articles/icy-rocks-workload-a-real-workload-for-the-android-platform) to represent a physics gaming workload. ACW is a further step forward in that it represents a wider set of application use-cases in the form of an Android workload. Intel's objective is to drive cross-platform ART compiler and runtime performance improvements on the latest and greatest versions of Android in order to improve user experience.

Downloads

The source code can be downloaded at https://github.com/android-workloads/JACWfA/.
Release 1.1 of the apk is available in the /bin folder.

Conclusion

Java* ACW for Android is a Java workload designed to stress real application components that in turn influence UX on Android Java-based applications. It is intended to help app developers write better apps, and Android VM engineers to better optimize existing libraries and Java runtime performance. Both contribute to better Android application performance and user experience. OEMs and customers are also encouraged to use this workload to gauge the Android software stack and CPU capabilities on mobile devices. Intel is using ACW 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.

About the authors

Aleksey Ignatenko is a Sr. Software Engineer in the Intel Software and Solutions Group (SSG), Systems Technologies & Optimizations (STO), Client Software Optimization (CSO). He focuses on Android workload development, optimizations in the Java runtimes and evolution of Android ecosystem.

Rahul Kandu is a Software Engineer in the Intel Software and Solutions Group (SSG), Systems Technologies & Optimizations (STO), Client Software Optimization (CSO). He focuses on Android performance and finds optimization opportunities to help Intel's performance in the Android ecosystem.

Acknowledgements (Alphabetical)

Dong-Yuan Chen, Chris Elford, Paul Hohensee, Serguei Katkov, Anil Kumar, Mark Mendell, Evgene Petrenko, Dmitry Petrochenko, Yevgeny Rouban, Desikan Saravanan, Dmitrii Silin, Vyacheslav Shakin, Kumar Shiv

Notice