Android application profiling

This section shows how to profile an Android application. Some examples are Here.

Profiling an Android application involves three steps:

1. Prepare an Android application.
2. Record profiling data.
3. Report profiling data.

Table of Contents

Prepare an Android application

Based on the profiling situation, we may need to customize the build script to generate an apk file specifically for profiling. Below are some suggestions.

1. If you want to profile a debug build of an application:

For the debug build type, Android studio sets android::debuggable="true" in AndroidManifest.xml, enables JNI checks and may not optimize C/C++ code. It can be profiled by simpleperf without any change.

2. If you want to profile a release build of an application:

For the release build type, Android studio sets android::debuggable="false" in AndroidManifest.xml, disables JNI checks and optimizes C/C++ code. However, security restrictions mean that only apps with android::debuggable set to true can be profiled. So simpleperf can only profile a release build under these three circumstances: If you are on a rooted device, you can profile any app.

If you are on Android >= Q, you can add profileableFromShell flag in AndroidManifest.xml, this makes a released app profileable by preinstalled profiling tools. In this case, simpleperf downloaded by adb will invoke simpleperf preinstalled in system image to profile the app.

```
<manifest ...>
    <application ...>
      <profileable android:shell="true" />
    </application>
</manifest>
```

If you are on Android >= O, we can use wrap.sh to profile a release build: Step 1: Add android::debuggable="true" in AndroidManifest.xml to enable profiling.

```
<manifest ...>
    <application android::debuggable="true" ...>
```

Step 2: Add wrap.sh in lib/ `arch` directories. wrap.sh runs the app without passing any debug flags to ART, so the app runs as a release app. wrap.sh can be done by adding the script below in app/build.gradle.

```
android {
    buildTypes {
        release {
            sourceSets {
                release {
                    resources {
                        srcDir {
                            "wrap_sh_lib_dir"
                        }
                    }
                }
            }
        }
    }
}

task createWrapShLibDir
    for (String abi : ["armeabi", "armeabi-v7a", "arm64-v8a", "x86", "x86_64"]) {
        def dir = new File("app/wrap_sh_lib_dir/lib/" + abi)
        dir.mkdirs()
        def wrapFile = new File(dir, "wrap.sh")
        wrapFile.withWriter { writer ->
            writer.write('#!/system/bin/sh\n\$@\n')
        }
    }
}
```

3. If you want to profile C/C++ code:

Android studio strips symbol table and debug info of native libraries in the apk. So the profiling results may contain unknown symbols or broken callgraphs. To fix this, we can pass app_profiler.py a directory containing unstripped native libraries via the -lib option. Usually the directory can be the path of your Android Studio project.

4. If you want to profile Java code:

On Android >= P, simpleperf supports profiling Java code, no matter whether it is executed by the interpreter, or JITed, or compiled into native instructions. So you don't need to do anything.

On Android O, simpleperf supports profiling Java code which is compiled into native instructions, and it also needs wrap.sh to use the compiled Java code. To compile Java code, we can pass app_profiler.py the --compile_java_code option.

On Android N, simpleperf supports profiling Java code that is compiled into native instructions. To compile java code, we can pass app_profiler.py the --compile_java_code option.

On Android <= M, simpleperf doesn't support profiling Java code.

Below I use application SimpleperfExampleWithNative. It builds an app-profiling.apk for profiling.

```
$ git clone https://android.googlesource.com/platform/system/extras
$ cd extras/simpleperf/demo
# Open SimpleperfExamplesWithNative project with Android studio, and build this pro
# successfully, otherwise the `./gradlew` command below will fail.
$ cd SimpleperfExampleWithNative

# On windows, use "gradlew" instead.
$ ./gradlew clean assemble
$ adb install -r app/build/outputs/apk/profiling/app-profiling.apk
```

Record and report profiling data

We can use app-profiler.py to profile Android applications.

```
# Cd to the directory of simpleperf scripts. Record perf.data.
# -p option selects the profiled app using its package name.
# --compile_java_code option compiles Java code into native instructions, which isn
# Android >= P.
# -a option selects the Activity to profile.
# -lib option gives the directory to find debug native libraries.
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative --co
    -a .MixActivity -lib path_of_SimpleperfExampleWithNative
```

This will collect profiling data in perf.data in the current directory, and related native binaries in binary_cache/.

Normally we need to use the app when profiling, otherwise we may record no samples. But in this case, the MixActivity starts a busy thread. So we don't need to use the app while profiling.

```
# Report perf.data in stdio interface.
$ python report.py
Cmdline: /data/data/com.example.simpleperf.simpleperfexamplewithnative/simpleperf re
Arch: arm64
Event: task-clock:u (type 1, config 1)
Samples: 10023
Event count: 10023000000

Overhead  Command     Pid   Tid   Shared Object                Symbol
```

```
27.04%    BusyThread  5703  5729  /system/lib64/libart.so    art::JniMethodStart(ar
25.87%    BusyThread  5703  5729  /system/lib64/libc.so      long StrToI<long, ...
...
```

report.py reports profiling data in stdio interface. If there are a lot of unknown symbols in the report, check here.

```
# Report perf.data in html interface.
$ python report_html.py

# Add source code and disassembly. Change the path of source_dirs if it not correct
$ python report_html.py --add_source_code --source_dirs path_of_SimpleperfExampleWi
      --add_disassembly
```

report_html.py generates report in report.html, and pops up a browser tab to show it.

Record and report call graph

We can record and report call graphs as below.

```
# Record dwarf based call graphs: add "-g" in the -r option.
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative \
        -r "-e task-clock:u -f 1000 --duration 10 -g" -lib path_of_SimpleperfExample

# Record stack frame based call graphs: add "--call-graph fp" in the -r option.
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative \
        -r "-e task-clock:u -f 1000 --duration 10 --call-graph fp" \
        -lib path_of_SimpleperfExampleWithNative

# Report call graphs in stdio interface.
$ python report.py -g

# Report call graphs in python Tk interface.
$ python report.py -g --gui

# Report call graphs in html interface.
$ python report_html.py

# Report call graphs in flamegraphs.
# On Windows, use inferno.bat instead of ./inferno.sh.
$ ./inferno.sh -sc
```

Report in html interface

We can use report_html.py to show profiling results in a web browser. report_html.py integrates chart statistics, sample table, flamegraphs, source code annotation and disassembly annotation. It is the recommended way to show reports.

```
$ python report_html.py
```

Show flamegraph

To show flamegraphs, we need to first record call graphs. Flamegraphs are shown by report_html.py in the “Flamegraph” tab. We can also use inferno to show flamegraphs directly.

```
# On Windows, use inferno.bat instead of ./inferno.sh.
$ ./inferno.sh -sc
```

We can also build flamegraphs using https://github.com/brendangregg/FlameGraph. Please make sure you have perl installed.

```
$ git clone https://github.com/brendangregg/FlameGraph.git
$ python report_sample.py --symfs binary_cache >out.perf
$ FlameGraph/stackcollapse-perf.pl out.perf >out.folded
$ FlameGraph/flamegraph.pl out.folded >a.svg
```

Record both on CPU time and off CPU time

We can record both on CPU time and off CPU time.

First check if trace-offcpu feature is supported on the device.

```
$ python run_simpleperf_on_device.py list --show-features
dwarf-based-call-graph
trace-offcpu
```

If trace-offcpu is supported, it will be shown in the feature list. Then we can try it.

```
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative -a .S
    -r "-g -e task-clock:u -f 1000 --duration 10 --trace-offcpu" \
    -lib path_of_SimpleperfExampleWithNative
$ python report_html.py --add_disassembly --add_source_code \
    --source_dirs path_of_SimpleperfExampleWithNative
```

Profile from launch

We can profile from launch of an application.

```
# Start simpleperf recording, then start the Activity to profile.
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative -a .M

# We can also start the Activity on the device manually.
# 1. Make sure the application isn't running or one of the recent apps.
# 2. Start simpleperf recording.
$ python app_profiler.py -p com.example.simpleperf.simpleperfexamplewithnative
# 3. Start the app manually on the device.
```

Control recording in application code

Simpleperf supports controlling recording from application code. Below is the workflow:

1. Run `api_profiler.py prepare` to enable simpleperf recording on a device. The script needs to run every time the device reboots.
2. Link simpleperf app_api code in the application. The app needs to be debuggable or profileableFromShell as described here. Then the app can use the api to start/pause/resume/stop recording. To start recording, the app_api forks a child process running simpleperf, and uses pipe files to send commands to the child process. After recording, a profiling data file is generated.
3. Run `api_profiler.py collect -p <package_name>` to collect profiling data files to host.

Examples are CppApi and JavaApi in demo.

Parse profiling data manually

We can also write python scripts to parse profiling data manually, by using simpleperf_report_lib.py. Examples are report_sample.py, report_html.py.