Simpleperf

Simpleperf is a native CPU profiling tool for Android. It can be used to profile both Android applications and native processes running on Android. It can profile both Java and C++ code on Android. The simpleperf executable can run on Android >=L, and Python scripts can be used on Android >= N.

Simpleperf is part of the Android Open Source Project. The source code is here. The latest document is here.

Table of Contents

Introduction

Simpleperf contains two parts: the simpleperf executable and Python scripts.

The simpleperf executable works similar to linux-tools-perf, but has some specific features for the Android profiling environment:

1. It collects more info in profiling data. Since the common workflow is “record on the device, and report on the host”, simpleperf not only collects samples in profiling data, but also collects needed symbols, device info and recording time.

2. It delivers new features for recording. a. When recording dwarf based call graph, simpleperf unwinds the stack before writing a sample to file. This is to save storage space on the device. b. Support tracing both on CPU time and off CPU time with --trace-offcpu option. c. Support recording callgraphs of JITed and interpreted Java code on Android >= P.

3. It relates closely to the Android platform. a. Is aware of Android environment, like using system properties to enable profiling, using run-as to profile in application's context. b. Supports reading

symbols and debug information from the .gnu_debugdata section, because system libraries are built with .gnu_debugdata section starting from Android O. c. Supports profiling shared libraries embedded in apk files. d. It uses the standard Android stack unwinder, so its results are consistent with all other Android tools.

4. It builds executables and shared libraries for different usages. a. Builds static executables on the device. Since static executables don't rely on any library, simpleperf executables can be pushed on any Android device and used to record profiling data. b. Builds executables on different hosts: Linux, Mac and Windows. These executables can be used to report on hosts. c. Builds report shared libraries on different hosts. The report library is used by different Python scripts to parse profiling data.

Detailed documentation for the simpleperf executable is here.

Python scripts are split into three parts according to their functions:

1. Scripts used for recording, like app_profiler.py, run_simpleperf_without_usb_connection.py.
2. Scripts used for reporting, like report.py, report_html.py, inferno.
3. Scripts used for parsing profiling data, like simpleperf_report_lib.py.

Detailed documentation for the Python scripts is here.

Tools in simpleperf

The simpleperf executables and Python scripts are located in simpleperf/ in ndk releases, and in system/extras/simpleperf/scripts/ in AOSP. Their functions are listed below.

bin/: contains executables and shared libraries.

bin/android/${arch}/simpleperf: static simpleperf executables used on the device.

bin/${host}/${arch}/simpleperf: simpleperf executables used on the host, only supports reporting.

bin/${host}/${arch}/libsimpleperf_report.${so/dylib/dll}: report shared libraries used on the host.

*.py, inferno: Python scripts used for recording and reporting.

Android application profiling

See android_application_profiling.md.

Android platform profiling

See android_platform_profiling.md.

Executable commands reference

See executable_commands_reference.md.

Scripts reference

See scripts_reference.md.

Answers to common issues

Why we suggest profiling on Android >= N devices?

```
1. Running on a device reflects a real running situation, so we suggest
profiling on real devices instead of emulators.
2. To profile Java code, we need ART running in oat mode, which is only
available >= L for rooted devices, and >= N for non-rooted devices.
3. Old Android versions are likely to be shipped with old kernels (< 3.18),
which may not support profiling features like recording dwarf based call graphs.
4. Old Android versions are likely to be shipped with Arm32 chips. In Arm32
mode, recording stack frame based call graphs doesn't work well.
```

Suggestions about recording call graphs

Below is our experiences of dwarf based call graphs and stack frame based call graphs.

dwarf based call graphs:

1. Need support of debug information in binaries.
2. Behave normally well on both ARM and ARM64, for both fully compiled Java code and C++ code.
3. Can only unwind 64K stack for each sample. So usually can't show complete flamegraph. But probably is enough for users to identify hot places.
4. Take more CPU time than stack frame based call graphs. So the sample frequency is suggested to be 1000 Hz. Thus at most 1000 samples per second.

stack frame based call graphs:

1. Need support of stack frame registers.
2. Don‘t work well on ARM. Because ARM is short of registers, and ARM and THUMB code have different stack frame registers. So the kernel can’t unwind user stack containing both ARM/THUMB code.
3. Also don‘t work well on fully compiled Java code on ARM64. Because the ART compiler doesn’t reserve stack frame registers.
4. Work well when profiling native programs on ARM64. One example is profiling surfacelinger. And usually shows complete flamegraph when it works well.
5. Take less CPU time than dwarf based call graphs. So the sample frequency can be 4000 Hz or higher.

So if you need to profile code on ARM or profile fully compiled Java code, dwarf based call graphs may be better. If you need to profile C++ code on ARM64, stack frame based call graphs may be better. After all, you can always try dwarf based call graph first, because it always produces reasonable results when given unstripped binaries properly. If it doesn't work well enough, then try stack frame based call graphs instead.

Simpleperf may need unstripped native binaries on the device to generate good dwarf based call graphs. It can be supported in two ways:

1. Use unstripped native binaries when building the apk, as here.
2. Download unstripped native libraries on device, as here.

How to solve missing symbols in report?

The simpleperf record command collects symbols on device in perf.data. But if the native libraries you use on device are stripped, this will result in a lot of unknown symbols in the report. A solution is to build binary_cache on host.

```
# Collect binaries needed by perf.data in binary_cache/.
$ python binary_cache_builder.py -lib NATIVE_LIB_DIR,...
```

The NATIVE_LIB_DIRs passed in -lib option are the directories containing unstripped native libraries on host. After running it, the native libraries containing symbol tables are collected in binary_cache/ for use when reporting.

```
$ python report.py --symfs binary_cache

# report_html.py searches binary_cache/ automatically, so you don't need to
# pass it any argument.
$ python report_html.py
```

Fix broken callchain stopped at C functions

When using dwarf based call graphs, simpleperf generates callchains during recording to save space. The debug information needed to unwind C functions is in .debug_frame section, which is usually stripped in native libraries in apks. To fix this, we can download unstripped version of native libraries on device, and ask simpleperf to use them when recording.

To use simpleperf directly:

```
# create native_libs dir on device, and push unstripped libs in it (nested dirs are
$ adb shell mkdir /data/local/tmp/native_libs
$ adb push <unstripped_dir>/*.so /data/local/tmp/native_libs
# run simpleperf record with --symfs option.
$ adb shell simpleperf record xxx --symfs /data/local/tmp/native_libs
```

To use app_profiler.py:

```
$ python app_profiler.py -lib <unstripped_dir>
```

Show annotated source code and disassembly

To show hot places at source code and instruction level, we need to show source code and disassembly with event count annotation. Simpleperf supports showing annotated source code and disassembly for C++ code and fully compiled Java code. Simpleperf supports two ways to do it:

1. Through report_html.py:

a. Generate perf.data and pull it on host. b. Generate binary_cache, containing elf files with debug information. Use -lib option to add libs with debug info. Do it with `binary_cache_builder.py -i perf.data -lib <dir_of_lib_with_debug_info>` . c. Use report_html.py to generate report.html with annotated source code and disassembly, as described here.

2. Through pprof.

a. Generate perf.data and binary_cache as above. b. Use pprof_proto_generator.py to generate pprof proto file. `pprof_proto_generator.py` . c. Use pprof to report a function with annotated source code, as described here.

Bugs and contribution

Bugs and feature requests can be submitted at http://github.com/android-ndk/ndk/issues. Patches can be uploaded to android-review.googlesource.com as here, or sent to email addresses listed here.

If you want to compile simpleperf C++ source code, follow below steps:

1. Download AOSP master branch as here.
2. Build simpleperf.

```
$ . build/envsetup.sh
$ lunch aosp_arm64-userdebug
$ mmma system/extras/simpleperf -j30
```

If built successfully, out/target/product/generic_arm64/system/bin/simpleperf is for ARM64, and out/target/product/generic_arm64/system/bin/simpleperf32 is for ARM.