Optimizing the software stack of a cosmic proportions cluster of multi-core machines

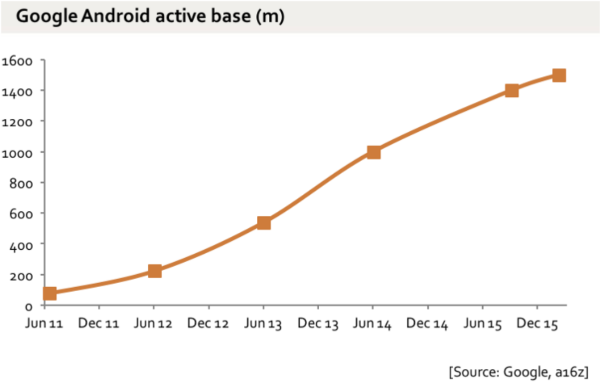
Sebastian Pop

SARC: Samsung Austin R&D Center

February 5, 2017

Android: a cosmic size cluster

- top500: $10M$ cores / $15.3MW$ [1] / US$273 million [2]
- Android devices: $\sim 6B$ cores [3] / $\sim 300MW$ [4] / $\sim$ US$0



[1] `https://www.top500.org/lists/2016/11`

[2] `https://en.wikipedia.org/wiki/Sunway_TaihuLight`

[3] 4 cores / device

[4] battery $13.2Wh = 4.4V * 3000mAh$, charging every 48 hours

Android Open Source Project (AOSP) Software Stack

- AOSP: common base for Android devices (+ customizations)
- C/C++ for the platform libraries, Java for user interface

ansic	22 MLoC	39%
cpp	13 MLoC	23%
java	10 MLoC	17%

- $\sim$ 80% execution cycles in C/C++, $\sim$ 20% in Java

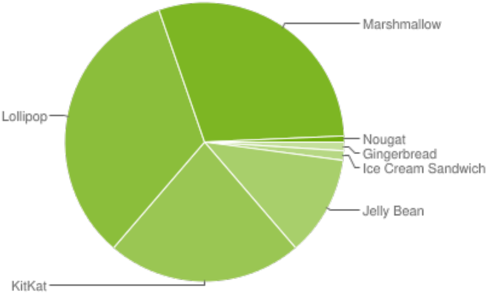
[5] Data collected during a 7-day period ending on January 9, 2017.

Android Open Source Project (AOSP) Software Stack

- AOSP: common base for Android devices (+ customizations)
- C/C++ for the platform libraries, Java for user interface

ansic	22 MLoC	39%
cpp	13 MLoC	23%
java	10 MLoC	17%

- $\sim$ 80% execution cycles in C/C++, $\sim$ 20% in Java
- release/updates/deprecation (5 $\sim$ 6 years)



[5]

[5] Data collected during a 7-day period ending on January 9, 2017.

Why Optimizing the Performance of Android?

Why bothering?

- the code of Android is cold (flat profile), full of branches
- there are few loops (image processing, compression, etc.)

[6] $\$0.12/kWh$, battery $13.2Wh = 4.4V * 3000mAh$, charging every 48 hours

Why Optimizing the Performance of Android?

Why bothering?

- the code of Android is cold (flat profile), full of branches
- there are few loops (image processing, compression, etc.)

Motivation:

- same code executed billions of time
- outer loop is outside the device
- profile how often code is in use
- variation over time following popularity of apps
- continuously monitor usage patterns
- tune code optimziations over time

$\$0.30$ / device / year $\longrightarrow$ $\$300M$ / billion devices / year [6]

[6] $\$0.12/kWh$, battery $13.2Wh = 4.4V * 3000mAh$, charging every 48 hours

Agenda

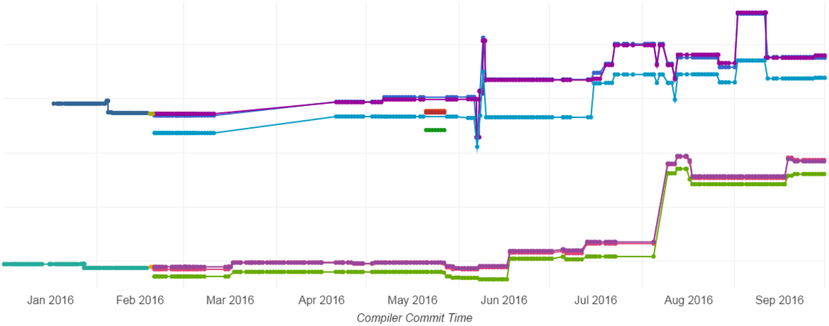
- Performance analysis: hot spots
- Improve performance of AOSP libraries
- Enable continuous profiling and optimizations (AutoFDO)
- Enable more secure execution environments (CFI)

Performance Analysis

- benchmarks: track performance over time (compiler/libraries)
- linux perf: profile of cycles (per function, hot-spots)
- valgrind: number of executed instructions (branches, R/W)
- static profiles: how many uses for a function

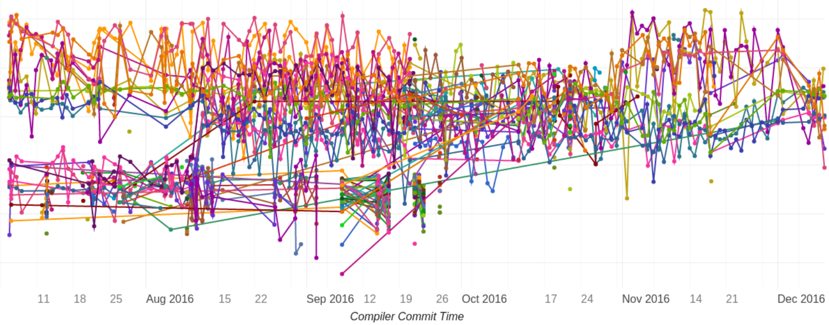
Benchmarks

track performance (compiler/libraries) over time



Benchmarks

on a real device (and a noisy benchmark...)



Linux Perf

Valgrind

Static Profile

- -flto: static call-graph, estimated frequencies per call

Improve performance of AOSP libraries

SARC contributions

- update Android NDK libc++, make it easy to keep updated
- 20x speedup of std::string.find() in libc++ and libstdc++
  need to port perf to memmem and strstr of bionic and glibc
- improve perf of shared_ptr in libc++
- improve perf of string to int value parsing in libc++

Benchmarking Standard Libraries

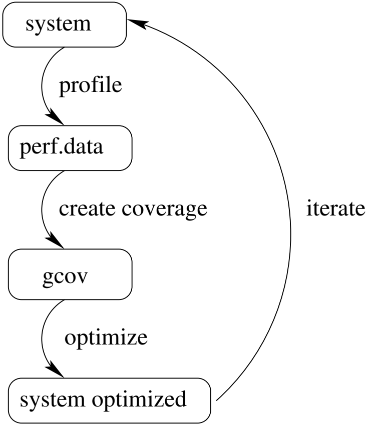
SARC contribution: std-benchmark[7]

- std-benchmark provides micro-benchmarks for functions in libc and C++ standard library
- detect room for improvement
  - compile with different compilers
  - link with different standard libraries
  - run on different machines: CPUs, architectures

[7] https://github.com/hiraditya/std-benchmark

AutoFDO: Feedback Directed Optimization

- linux-perf extracts profiles of running systems
- little to no overhead [8]
- coverage (basic block frequencies) from dynamic profiles
- continuous profiling and tuning of optimizations



[8] Google Wide Profiling: A Continuous Profiling Infrastructure for Data Centers, IEEE Micro (2010)

AutoFDO: Example

AutoFDO: Code Optimizations

- better inlining [9], devirtualization, function instantiation
- hot/cold code placement
- register allocation, jump-threading, etc.

[9] Lightweight Feedback-Directed Cross-Module Optimization, CGO 2010

AutoFDO: More Precise Coverage

- Intel-LBR (Last Branch Record): last 16 taken branches
- provides more precise basic block execution frequency
- how do we do this on ARM?

ARM-ETM: Embedded Trace Macrocell

- ARM-ETM: records execution traces (for debug)
- dedicated circular buffer 1 to 3MB ($\sim 10^5$ branches/MB)
- no overhead
- support in Linux kernel by Mathieu Poirier (Linaro)
- next android kernel linux-4.9 will support ARM-ETM

- **SARC contribution**: how to use ARM-ETM for AutoFDO
  - perf-inject translates execution traces to LBR events
  - patch similar to perf-inject for Intel Process Trace

AutoFDO: with ARM-ETM

sort.c

gcc -O3 -g sort.c -o sort.exe

sort.exe

perf record -e cs_etm/@20070000.etr/u --per-thread ./sort.exe

perf.data — contains ETM execution traces

perf inject -i perf.data -o inject.data --itrace — compile ETM to LBR

inject.data — contains LBR events

create_gcov --binary=sort.exe --profile=inject.data --gcov=sort.gcov

sort.gcov

gcc -O3 -fauto-profile=sort.gcov sort.c -o sort-autofdo.exe

sort-autofdo.exe

From Dynamic Profiles to Power Usage

- traditionally, per app battery usage (ammeter on wire) [10]
- more accurate picture with linux-perf profiles:
  - profiles from the field: real world use-cases
  - merge together different profiles
  - compute code execution frequency
  - power consumption estimation per line of code

[10] An Analysis of Power Consumption in a Smartphone, USENIX'10

Towards more secure devices

- Control Flow Integrity (CFI): 2% overhead [11]
- to enable on Android: need to further reduce its cost

[11] Enforcing Forward-Edge Control-Flow Integrity in GCC&LLVM, USENIX'14