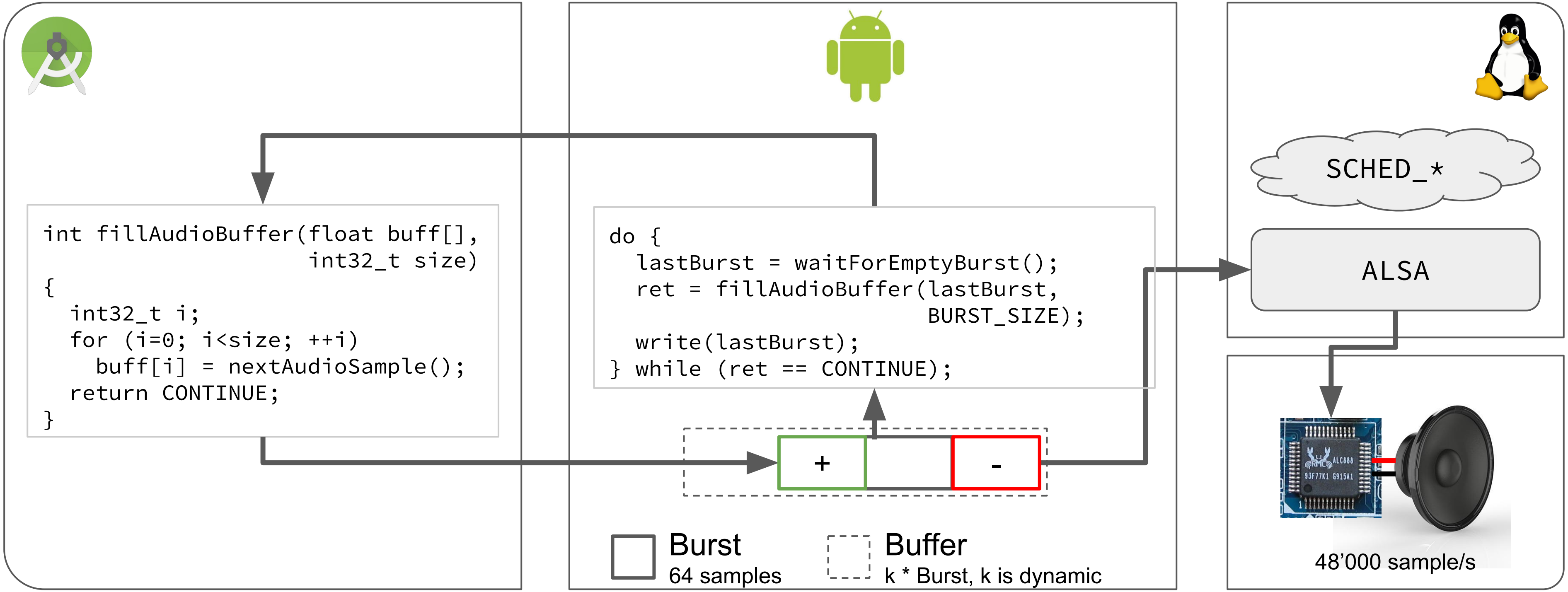
Experimenting with the Android Audio Pipeline and SCHED_DEADLINE

OSPM 2018

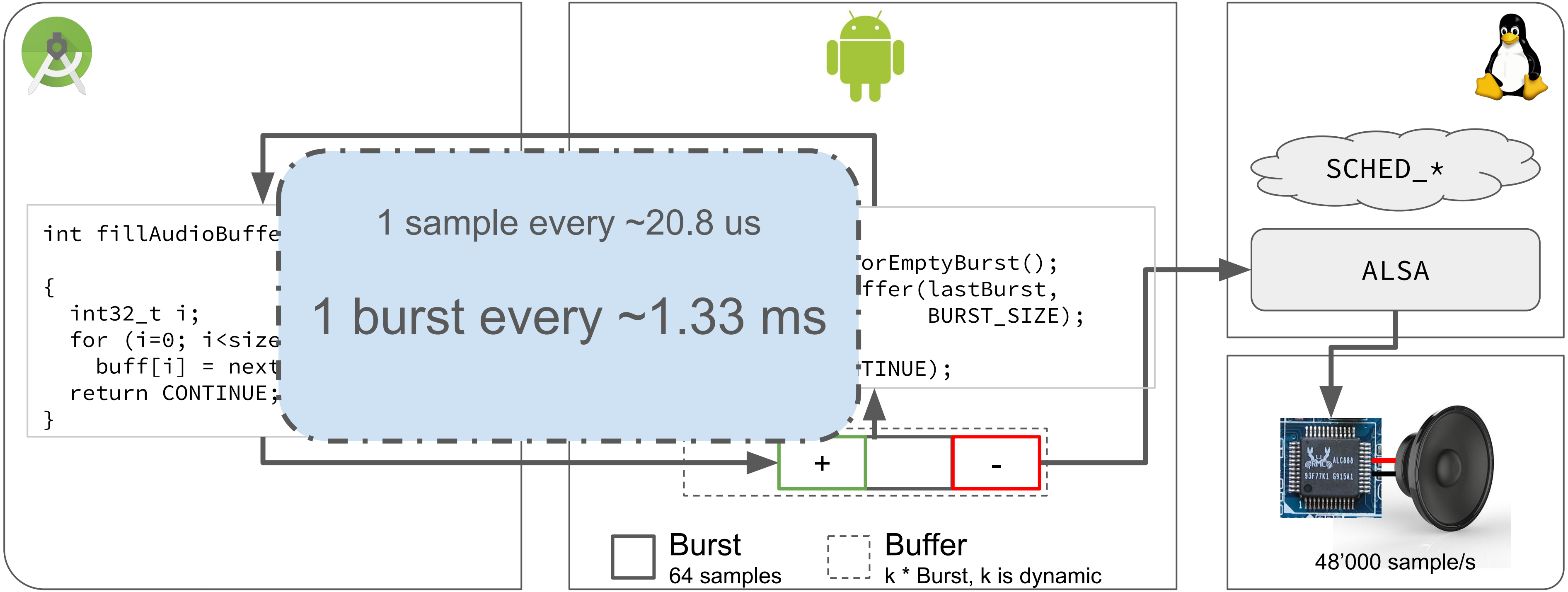
Alessio Balsini - Arm, Scuola Superiore Sant'Anna
With the contribution of:
Joel Fernandes and Phil Burk - Google

Sant'Anna
Scuola Universitaria Superiore Pisa

Software Architecture: Audio Pipeline Simplified



Software Architecture: Audio Pipeline Simplified



Requirements

Power efficient

- Forcing the maximum frequency is not always good (e.g., power consumption, thermal throttling, current pumping), how to find the **smallest**, but **sufficient OPP**?

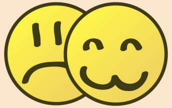
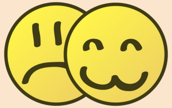
Low-latency

- Best-effort scheduling may result in underruns/overruns: audio glitches
- The buffer must be full, so, the bigger the buffer, the bigger the latency

Reactive to workload changes

- The **workload** of an audio **varies** a lot (e.g., different number of notes), so load increase → quick CPU frequency increase!

Alternative Solutions

	Power Efficiency	Reactiveness to Workload	Latency
SCHED_{RT, DL} max frequency (*)	?		
SCHED_RT, WALT or PELT (**)			
SCHED_DL adaptive BW			

(*) Race to Idle?

(**) With not (yet) mainline solutions, reactiveness and latency can be mitigated, but it's still required to specify from userspace either the utilization or frequency clamping mechanisms.

SCHED_DEADLINE with Adaptive Bandwidth

SCHED_DEADLINE is good because:

- Fits periodic tasks
- Tailored for deadline tasks

How to choose the parameters?

1. The callback is activated every time a burst is available (**period** = 1.33 ms)
2. A burst must be filled before another is consumed (**deadline** = 1.33 ms)
3. The **runtime** is unknown! But we have past execution **statistics** and the **application** can **hint** the system (adaptive loop)

```
int fillAudioBuffer(float buff[],
                    int32_t size)
{
  int32_t i;
  for (i=0; i<size; ++i)
    buff[i] = genNextAudioSample();
  return CONTINUE;
}
```

```
setAppWorkUnits(WU_new);
```

```
void setAppWorkUnits(WU_new) {
  /* estimate runtime and  *
   * do sched_setattr(...) */
}
```

```
do {
  lastBurst = waitForEmptyBurst();
  ret = fillAudioBuffer(lastBurst,
                        BURST_SIZE);
  updateRuntimeStatistics(WU_curr);
  write(lastBurst)
} while (ret == CONTINUE);
```

Experimental Setup

- **Device**: fake battery Google Pixel 2
- **Android**: AOSP, version P (master of)
- **Kernel**: MSM walleye 4.4 with latest mainline (and not) DL patches
  `(android.googlesource.com/kernel/msm)`
- **Power meter**: ACME Cape RevB
- **Workload generator**: audio synthesizer emulator, that generates audio workload and stores statistics `(github.com/google/synthmark)`
- **Test automation toolkit**: LISA `(github.com/ARM-software/lisa)`

The experiments are run on **big** CPUs

Experimental Results: Notes Switching

```
walleye:/ # /data/synthmark -s60 -tl -n5 -N140 -Rs

  msecPerBurst    =

Latency is 128 frames = 2.66667 msec at burst siz
walleye:/ # _
```

Audio Latency

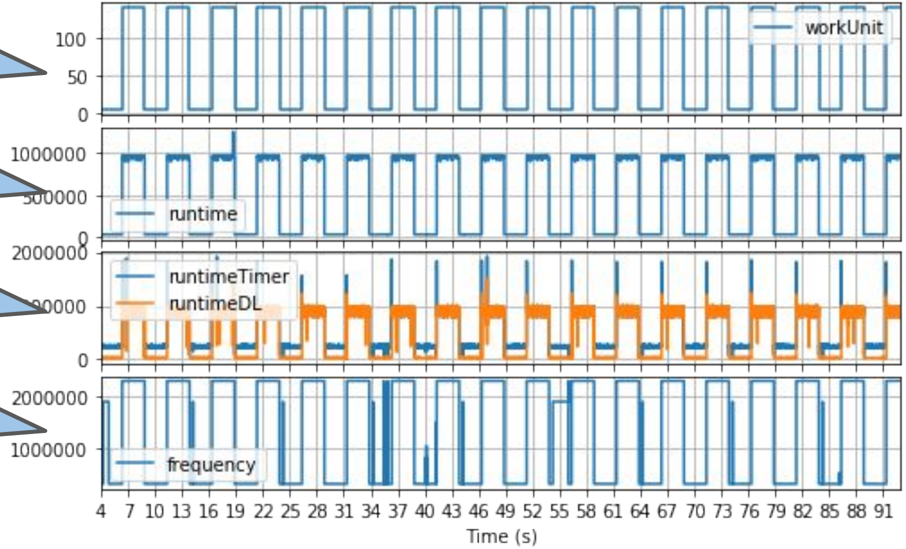
Time to generate one audio burst (histogram)

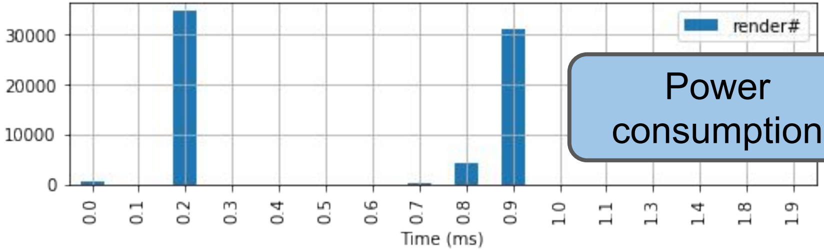
Notes played

Predicted runtime

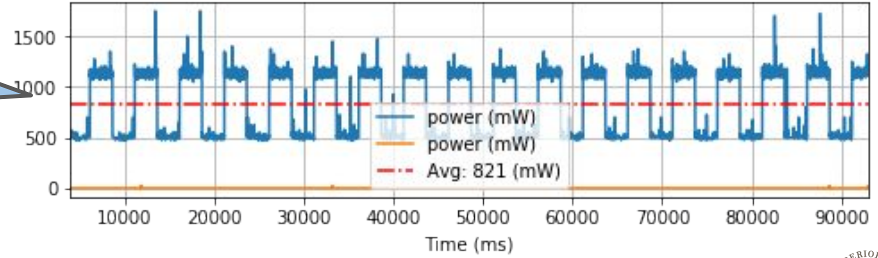
Measured runtime

CPU frequency



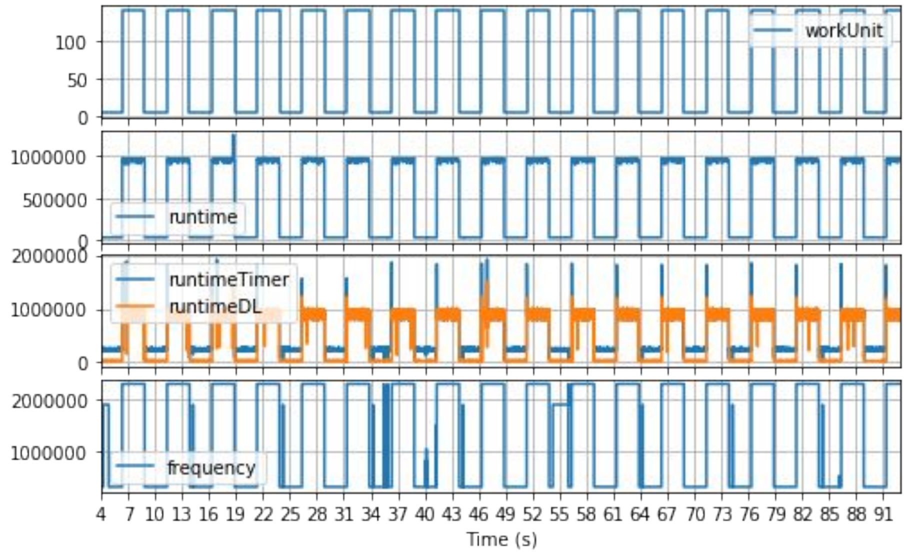


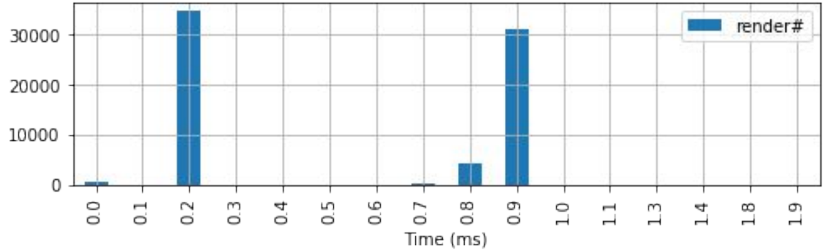
Power consumption

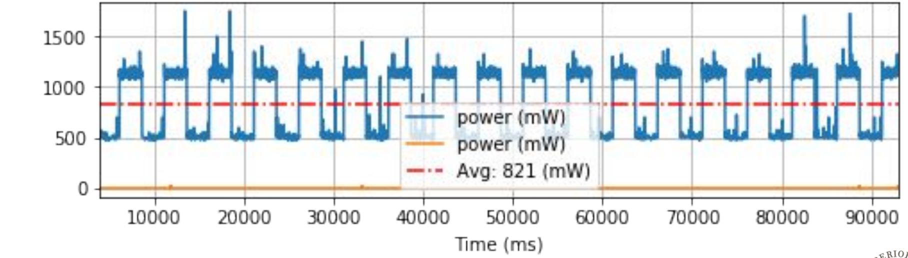


Experimental Results: Notes Switching

```
walleye:/ # /data/synthmark -s60 -tl -n5 -N140 -Rs

--- SynthMark V1.12 ---
  test name       = LatencyMark
  numVoices       =      5
  numVoicesHigh   =    140
  framesPerBurst  =     64
  msecPerBurst    =   1.33

Latency is 128 frames = 2.66667 msec at burst size 64 frames
walleye:/ # _
```

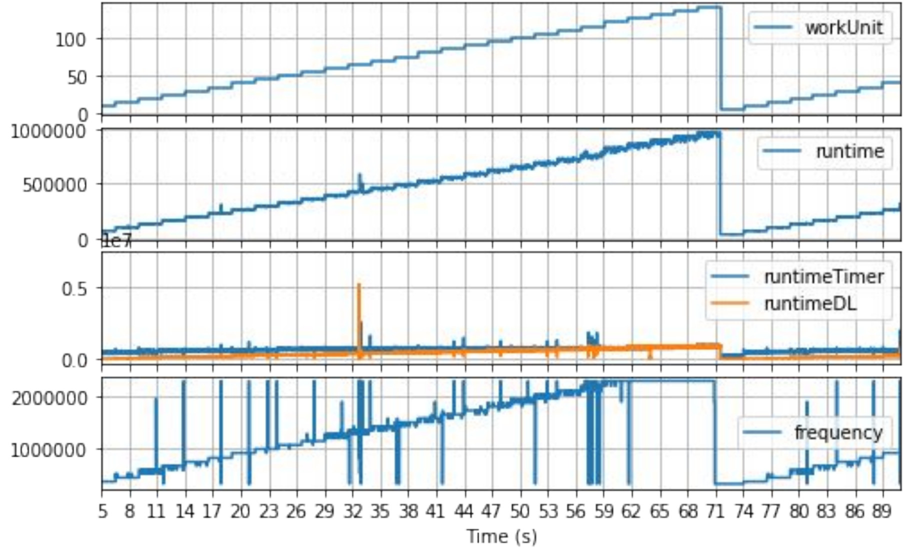


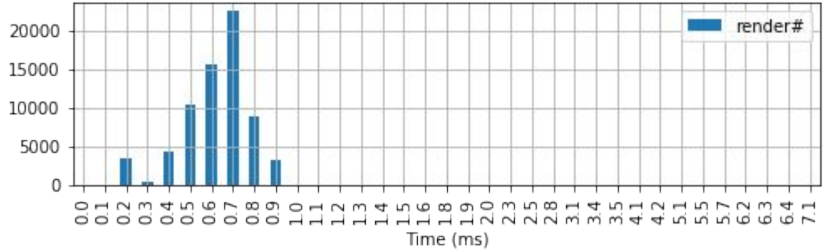


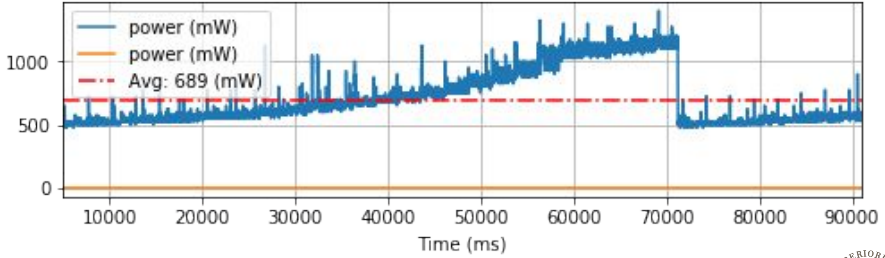


Experimental Results: Linear Increment

```
walleye:/ # /data/synthmark -s60 -tl -n5 -N140 -Rl

--- SynthMark V1.12 ---
  test name      = LatencyMark
  numVoices      =      5
  numVoicesHigh  =    140
  framesPerBurst =     64
  msecPerBurst   =   1.33

Latency is 128 frames = 2.66667 msec at burst size 64 frames
walleye:/ # _
```

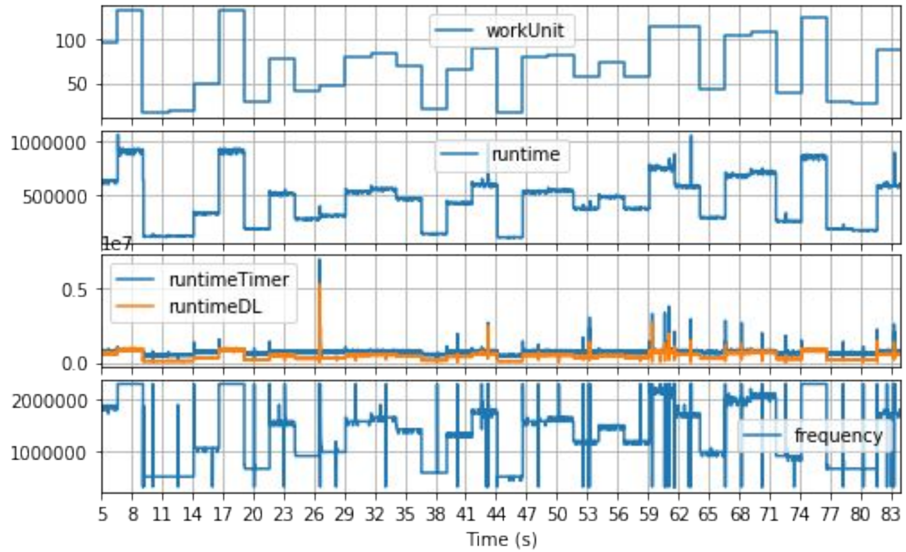


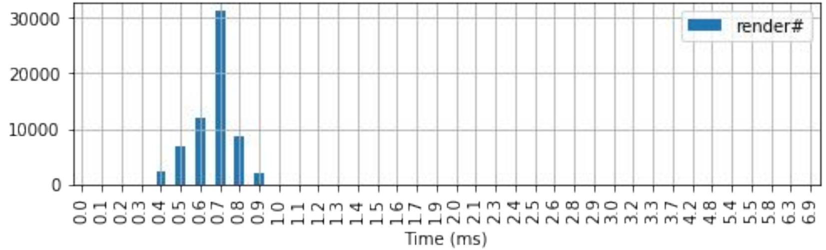


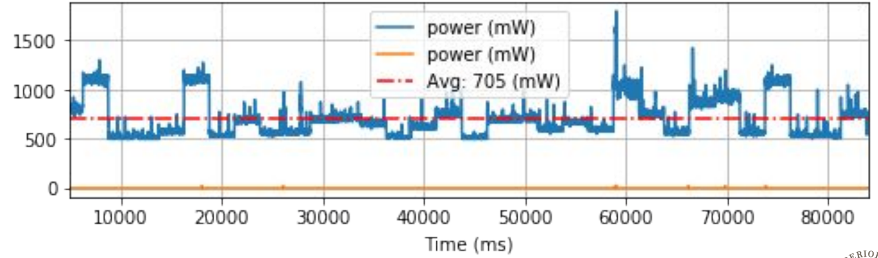


Experimental Results: Random

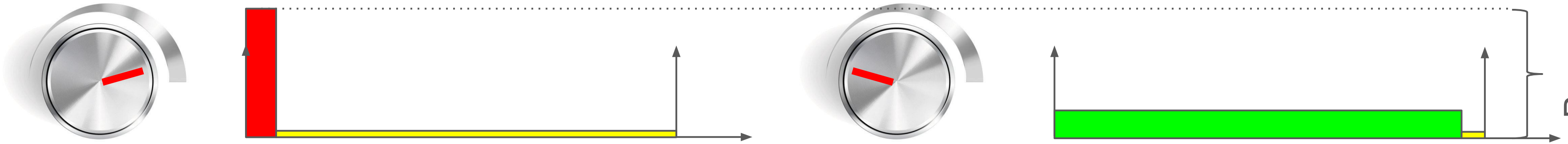
```
walleye:/ # /data/synthmark -s60 -tl -n5 -N140 -Rr

--- SynthMark V1.12 ---
  test name       = LatencyMark
  numVoices       =      5
  numVoicesHigh   =    140
  framesPerBurst  =     64
  msecPerBurst    =   1.33

Latency is 128 frames = 2.66667 msec at burst size 64 frames
walleye:/ # _
```







To Race or not to Race (to Idle)?

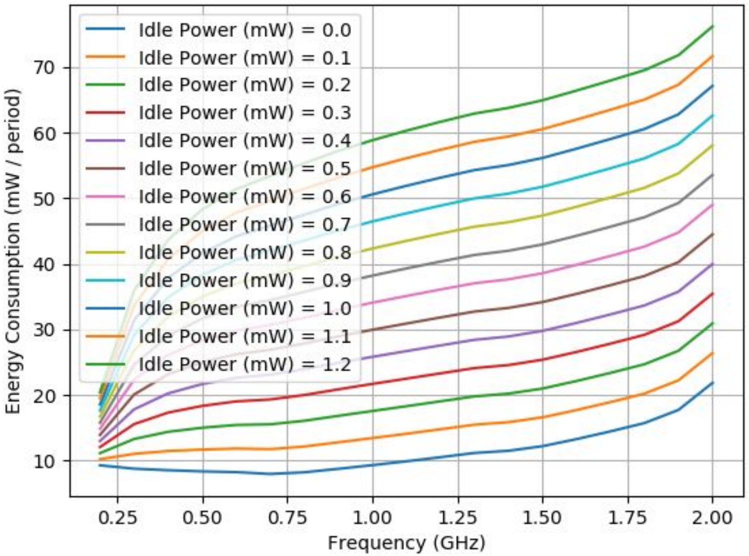


Frequency

Race to Idle

Frequency

GRUB-PA



Odroid XU3 board (Samsung Exynos-5422)

With "old" processors (Exynos-5422) is better not to waste power doing nothing.

On Pixel 2 (Qualcomm Snapdragon 835), the clock gating is very efficient → race to idle is fine.

So? Device specific!

Needs

Android ⇆ App

- App → Android: API to notify that the App load is changing (to update of the DL parameters ASAP)
- Android/Kernel → App: API to notify the App of the available system capacity

Android ⇆ Kernel

- API to measure the execution time (a kind of se.sum_exec_runtime), frequency + capacity scaled.
  In this project implemented with custom `sched_getattr()` flag: returns `dl_se.runtime`.
  What about `CLOCK_THREAD_CPUTIME_SCALED_ID`, accessible with `clock_gettime()`?

Kernel

- Capacity-aware scheduling: migrate to a big CPU if its requirements do not fit the LITTLE

Other Ideas

Export the available DL bandwidth

- Not blindly ask the admission controller
- Some applications can adapt their workload according to the system availability

The period of the task is not the period of CBS: synchronization callback!

- Maybe with a new flag for `sched_setattr()`

Multilevel SCHED_DEADLINE

- Split SCHED_DL into soft vs hard deadline tasks, the bandwidth is guaranteed only to hard tasks, best effort for the others (but still deadline based)

Thank you!

arm

Alessio Balsini - Arm, Scuola Superiore Sant’Anna
alessio.balsini@{arm.com, santannapisa.it}
With the contribution of:
Joel Fernandes and Phil Burk - Google