SchedTune: Capacity Clamping

Why is needed and which API should we use?

Patrick Bellasi

<patrick.bellasi@arm.com>

Agenda

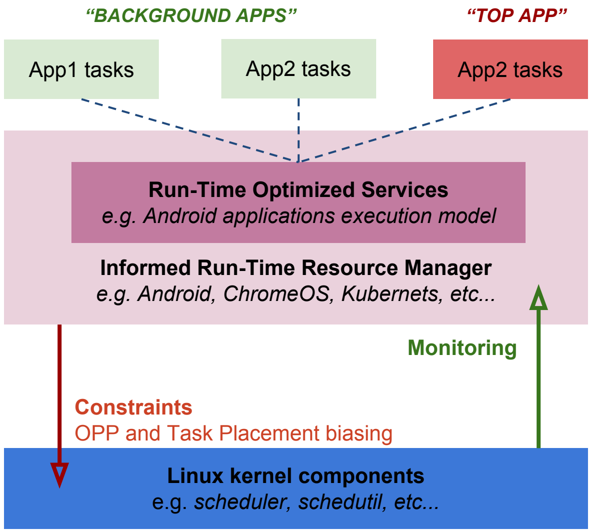
- Introduction
  problem and goals, Android use-case
- Proposal
  new concepts, evaluated alternatives and supposed strengths
- Discussion
  walking-through the main controversial points

- On-demand contents
  implementation details, validation, future works, ...

Introduction

What is the problem on hand?

Feed ***context aware*** *information about* ***tasks requirements***
from ***informed run-times*** *to kernel-space*
to *<u>improve existing decision policies</u>* *for* ***OPPs selections*** *and* ***tasks placement***



Informed run-time **managed** applications

- resources partitioning
  how many and which CPUs can an app use?
- apps/tasks priorities tuning
  what is the priority of certain task?
- defined optimization goals
  energy-saving vs performance-boosting

Manage **transient** configurations

- which app is now more important?
- Boost performances on certain events
  e.g. touchboost, app startup

Introduction

The Android Use-Case

A set of concepts have been evaluated during the Pixel's tuning exercise

- **boost** TA's tasks: prefer more capable CPUs and run faster than required
  tasks pinning is not possible for boosted apps: we still want all CPUs when available (i.e. best effort)
  tasks reported as small by PELT can still benefit from a faster completion time (i.e. run at higher OPPs)
- **prefer_idle** for latency sensitive tasks
  while still being energy-efficient when idle CPUs are not available at wakeup time
- experiments using "**negative boosting**"
  controlled performance degradation (i.e. RTM reduces the resources => apps automatically adapt)

Energy-efficiency and **Low Latencies** are both required for different class of tasks

- depending on task status, e.g. TA vs BG

	Neg Boosting	**No Boost**	**Boosting**
Energy Efficiency	BG	BG / SYS_BG	FG (non TA)
Lower Latency	Camera	FG	TA

Proposal Concepts Mapping on Existing and New interfaces

Original Concepts	Mapping within the CPU Controller
Boost value	Using the existing **cpu.shares** attribute - by default tasks have a 1024 share - boosted tasks gets a share >1024 (more CPU time to run) - negative boosted tasks gets <1024 (less CPU time to run)
OPP biasing	Add a new **cpu.min_capacity** attribute Tasks in the group are granted to be scheduled on a CPU which provides at least the required minimum capacity
Negative boosting	Add a new **cpu.max_capacity** attribute Tasks in the group are never scheduled on a cpu with CPU capacity higher that this value (at least while they are alone on that CPU)
CPU selection and prefer_idle	The **cpu.shares** value can be used as a "flag" to know when a task is boosted e.g. is cpu.shares > 1024 (or threshold) we look for an idle CPU The **cpu.min_capacity** can also bias the selection of a big CPU The **cpu.max_capacity** can also bias the selection of a LITTLE CPU
Latencies reduction	Tasks with higher **cpu.shares** value are entitled more CPU time and this turns out to give them better chances to get scheduled by preempting other tasks with lower shares. **NOTE:** the CPU bandwidth not consumed by high **cpu.shares** value tasks is still available for tasks with lower shares.

Concept already available (Boost value)

CPU utilization clamping (OPP biasing, Negative boosting)
https://lkml.org/lkml/2017/2/28/355

Never poster on LKML

Task Placement (CPU selection and prefer_idle)

Performance Boosting (Latencies reduction)

Proposal

What alternative ways have been considered?

Existing APIs seems to be limited:

- task's affinity: enforce scheduling from user-space, too much aggressive for TOP_APP
- tasks priorities: mainly used to partition CPU time among RUNNABLE tasks
- cpusets and cpu controller: are the most promising
  but they are not "feature-complete" to support biasing of OPP selection and tasks placement

we are looking for a "suitable extension"

to bias OPP selection and tasks placement

Initial solution[1] was proposing a complete new CGroup controller

- Tejun complained about compliance with CGroups v2
- PaulT and Tejun suggested to extend the cpu controller[2]
  *to get also a more **consistent view** about the "**allocation of the CPU resource**"*

[1] https://lkml.org/lkml/2016/10/27/503
[2] https://lkml.org/lkml/2016/11/25/342

Proposal

Why the current proposal has been chosen?

Main benefits we thinks are:

- simple interface towards “informed run-time” with “context aware” info
  which already uses CGroups to allocate resources to group or tasks (i.e. apps)

- builds biasing on top of existing policies
  for both OPP biasing (current proposal) as well as task placement (with a future extension)

- enable the CPU controller to enforce min/max **computational bandwidth**
  not only time **computational time** like what we have now

- by default, it does not enforce any new/different behavior
  it just open to opportunistic tuning of CFS tasks whenever necessary

- it has almost negligible run-time overhead
  mainly defined by the complexity of a couple or RBTree operations

ARM

Discussion

Main controversial points (1/3)

Does the concepts of **capacity_min** makes sense to have?

- doubts about being required just because of **other bits being suboptimal**
  PELT under-estimating task demands, being slow, ...
  {cfs,rt}_{period,runtime}_us enforce only time, not actual computational bandwidth[1]
- is capacity_min really useful to define an energy-vs-performance tradeoff?
  should be better a dedicated concept of per-task “boost value”?
- current implementation targets both FAIR and RT classes
  does it makes sense to use it as a “best effort” extension to cfs/rt bandwidth controllers?
- it’s an API to “require for more”, thus potentially exploitable by user-space apps
  should require special permissions to be used?

[1] potentially, maybe they can be extended to be freq/arch invariant

Discussion

Main controversial points (2/3)

What is the proper **semantic for capacity_{max,min}**?

- how they should be inherited?
  child geting same value of parent, could that work?

- how they should be restricted walking down a CGroup hierarchy?
  capacity_max can only be **smaller**: matches bandwidth controllers delegation model
  capacity_min can only be **bigger**
    the rough idea is for contained tubgroups to not affect parent performances
    this is the most controvertial sematinc... any good reason to do the opposite?

- is "capacity" a sufficiently generic concept across different platforms?
  is it not normalized in any way between architectures?

ARM

Discussion

Main controversial points (3/3)

Is it appropriate to use CGroups as a **primary interface**?

- capacity_{min,max} are not limits on **countable units** of a specific resource
  this is more likely an **attribute range restriction** controller
  is it ok to use a "property restriction model" similar to the taskaffinity/cpusets one?

- apps should be allowed to set capacity_{min,max} without CGroups
  do we really want to expose directly such an interface to apps?
  does it makes sense to have apps, potentially non priviledged, using capacity_{min,max}?
  which restrictions should be put in place?

- what can be a suitable "primary interface"?
  Joel's proposal: extend the **prlimit** API, can it works for capacity_min?
  what's the most convenient "regulare API"?

ARM

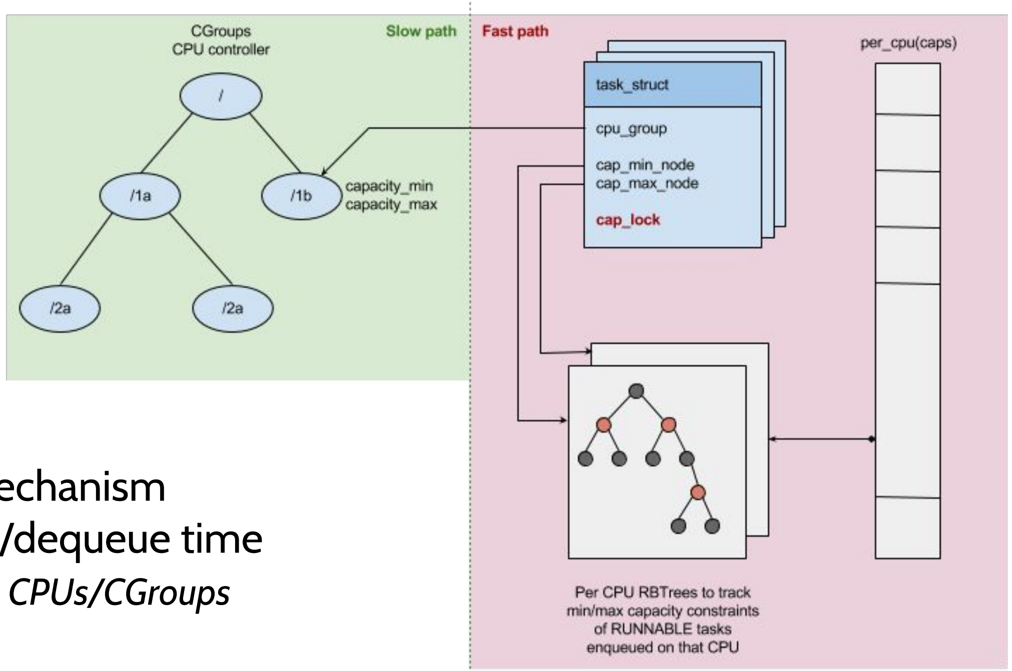
Backup Slides

SchedTune v3

Implementation Details

CPUs keep track of capacity constraints

- for all RUNNABLE tasks
- using RBTrees to keep **task_struct** ordered



Tasks ordered based on capacity constraints enforced by their CGroups

- simple accounting and aggregation mechanism
- insertion/removal ops just at enqueue/dequeue time

  free support for tasks migrations between CPUs/CGroups

Main features

- capacity clamping tracked by the core scheduler

  support for both FAIR and RT tasks
- No limitations on number of “boost groups”

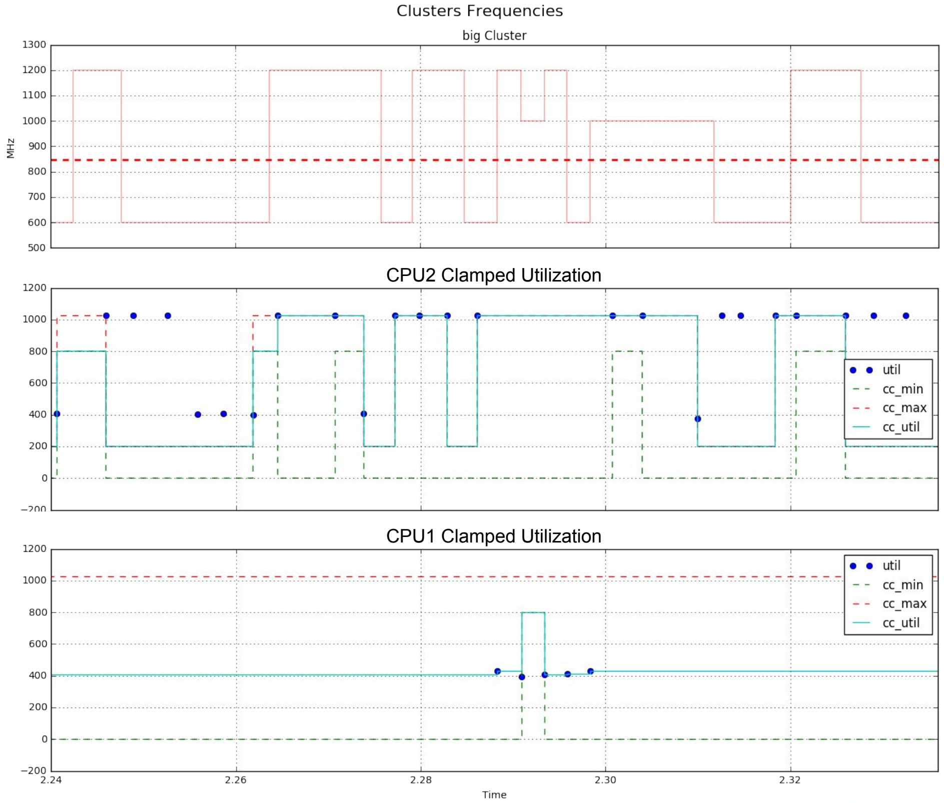
SchedTune v3: Capacity Clamping Validation

Functional validation performed on JUNO R2 boards

- using this rt-app synthetic scenario

```
10x10%   background tasks
         capacity_max=20%
         cpumax=0x4

 1x10%   top-app task
         capacity_min=80%
         cpumax=0x4
```



[1] https://gist.github.com/derkling/5409bae8e358ba92da3a5a10921f6d59

SchedTune v3: Shares Benefits on Latencies

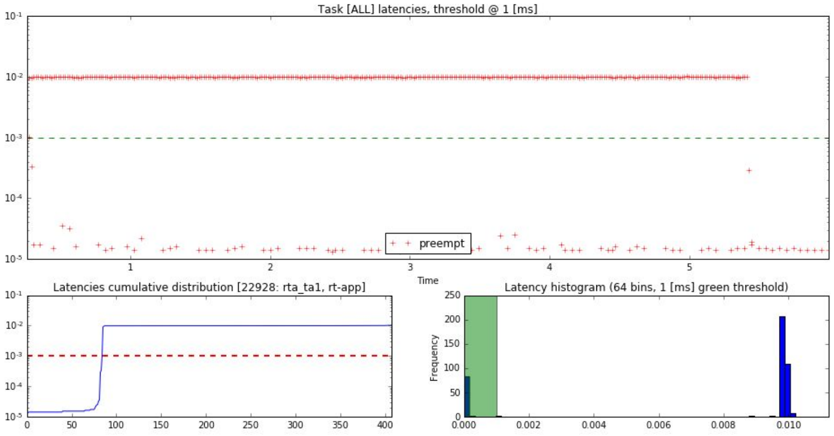


Fig.1 - Shares: BG=1024, TA=1024

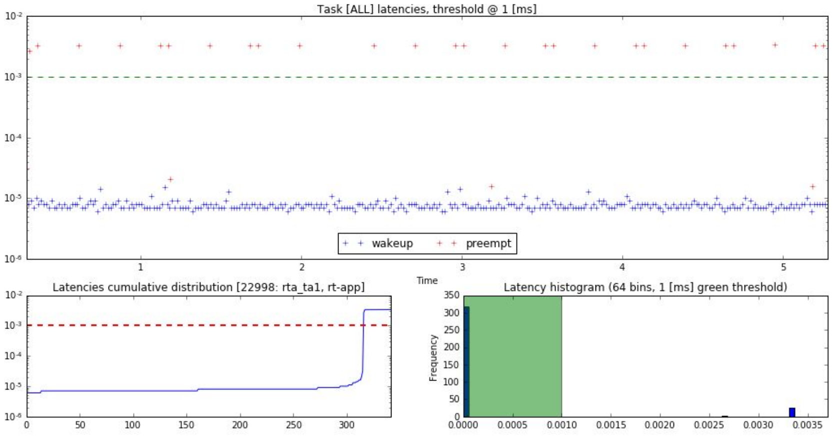


Fig.2 - Shares: BG=2, TA=1024

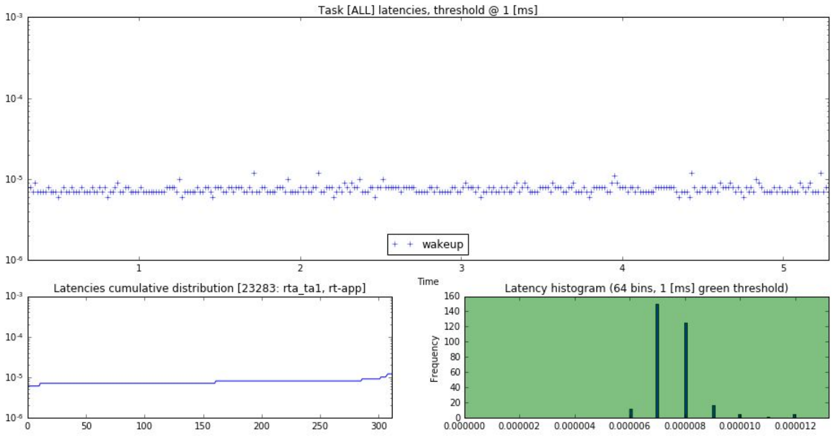


Fig.3 - Shares: BG=2, TA=10K

SchedTune: Design Goals

Provide a simple, central tunable for

energy saving vs performance boosting

Bias **OPP selection** and **tasks placement**

- provide schedutil with behaviours similar to other governors
  e.g. interactive, performance
- support EAS to trade-off **energy saving** for **performance boosting**

Fosters the collection of sensible information from **informed run-times**

- to support better task scheduling decisions
- by providing a simple yet effective API to middleware like Android

ARM

SchedTune: Current Status (i.e. what's in use)

RFC v2 posted on LKML [1]

- supporting only OPP boosting but based on schedutil integration

Full solution available in ACK v3.18 [2]

- supporting task biasing via EAS integration
  - *in **find_best_target()** for **!is_big_little** targets*
- small refinements to support either PELT or WALT utilization signal
- using additional attribute to better support latency sensitive tasks

Further fixes and improvements in MSM v3.18 [3]

- available in partner's msm-google kernel tree
- improved performance index definition

[1] https://lkml.org/lkml/2016/10/27/503
[2] https://android.googlesource.com/kernel/common/+/android-3.18
[3] https://android.googlesource.com/kernel/msm/+/android-msm-marlin-3.18-nougat-mr1-eas-experimental

SchedTune: Improved Performance Index

Performance index discounting for **potential delay sources**

Perf_idx = SpeedUp_idx **– Delay_idx**

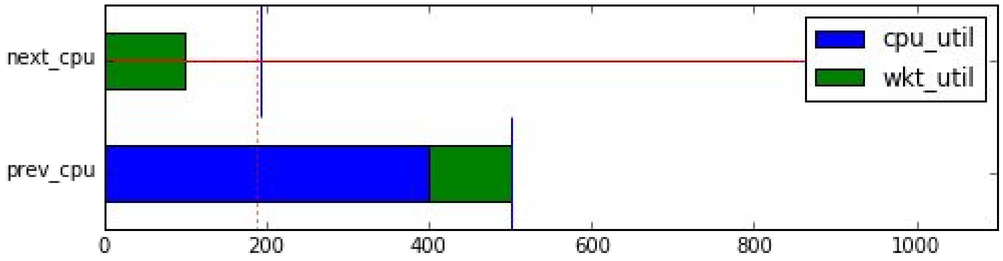
- estimate of "how fast" the task will run

  SpeedUp_idx = cpu_boosted_capacity – task_util

- discount all the latency treats (e.g. co-scheduling, Hi-Prio tasks, blocked-load, IRQ pressure, etc.)

  Delay_idx = 1024 * (cpu_util - task_util) / cpu_util [1]

10% tasks waking up, 10% boost (~90 utilization margin)



Next_cpu preferred depending on:
- prev_cpu utilization and blocked load
- boosted CPU's capacity

[1] cpu_util includes task's utilization

SchedTune: Main Complains from LKML/LPC [1]

Introduction of a new CGroup controller

- the boost value is **affecting** the availability of **CPU's bandwidth**
- Tejun&PaulT proposed to integrate this concept into the **existing CPU controller**

    this should support a more coherent view on what is the status of the CPU resource

Enforcing (by design) a "flat hierarchy" of boosted tasks

- a flat hierarchy does not match the expected **"generic behaviors" for CGroup interface**
- such a controller cannot be easily extended to **support CGroup v2** configuration

The request for a single knob has been kind-of demoted

- some implementation details currently do not allow to **grant the required boost** values
- boosting support is really required only for **mid-to-big deltas**

    e.g. small tasks with big boosting, but not the big tasks with small deltas
    a threshold based implementation could be potentially good enough

[1] https://lkml.org/lkml/2016/11/25/342

ARM

SchedTune v3: Works in Progress

Complete task placement biasing

- remap **prefer_idle** to a suitable check condition on **cpu.shares** value
- the performance index will not be added in the first instance

Integrate v3 (possibly beside v2) in EAS r1.3 for ACK 4.4

Complete the AOSP userspace integration

- refactor/cleanup current sched_policy[1]
- extends full task classes to cpuctl
  - BACKGROUND
  - SYSTEM_BACKGROUND
  - FOREGROUND
  - TOP_APP
- update both cpuctl and cpuset at each policy setting/updating

- **android/platform/system/core**
  - rootdir/init.rc
  - libcutils/sched_policy.c
- **android/platform/frameworks/av**
  - media/audioserver/audioserver.rc
  - media/mediaserver/mediaserver.rc
  - camera/cameraserver/cameraserver.rc
- **android/platform/frameworks/base**
  - services/core/java/com/android/server/UiThread.java
  - cmds/bootanimation/bootanim.rc
  - core/java/android/os/Process.java

[1] https://android.googlesource.com/platform/system/core/+/master/libcutils/sched_policy.cpp

ARM

SchedTune v3: Future Advanced Topics

Experiment with CFS bandwidth controller

- investigate the possibility to replace the usage of cpusets with a proper and more complete configuration of the CPU bandwidth controller
- should optimize parallelization of background tasks, especially when there are not foreground and/or top apps running

Using per-app CGroups instead of task classes

- this is expected to reduce overheads related to moving tasks around
- better match the most "classical" usage of the CGroup interface, i.e.

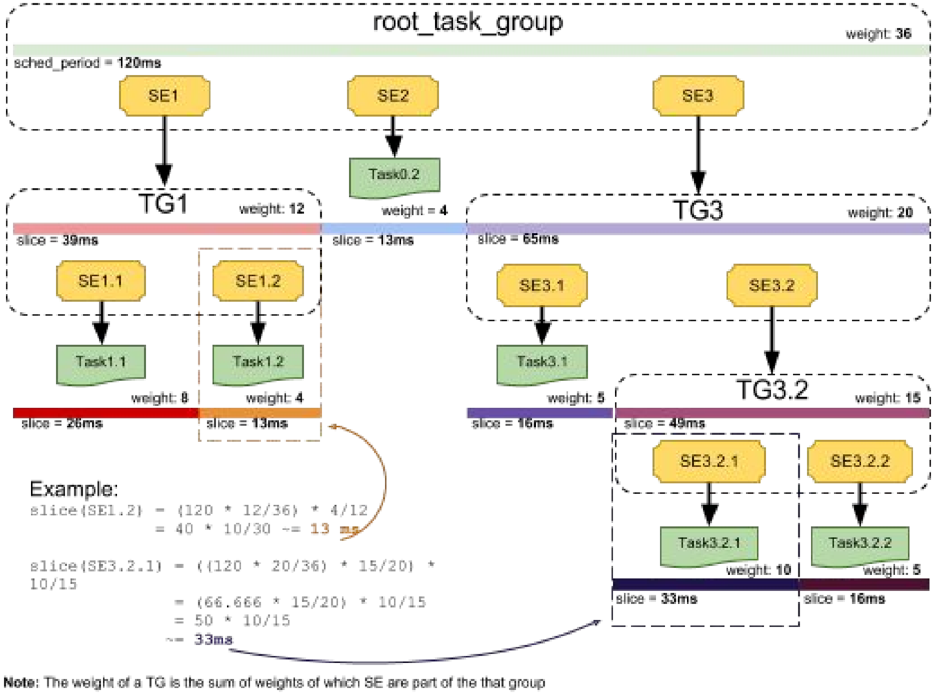
"Organize Once and Control [1]"

[1] http://lxr.linux.no/linux+v4.10.1/Documentation/cgroup-v2.txt#L361

ARM

SchedTune v3: How Shares Works?

Similarly to how SE's priority defines the "weight" of a TG, and thus its slice time



- Used to repartition the **scheduling latency** (SL)
  - **/proc/sys/kernel/sched_latency_ns**
  - 10ms by default in AOSP
- A quota of SL, proportional to its share, is assigned to each SE
  - never smaller than: **sched_min_granularity_ns**
-

