Google

Can mainline Linux run on Android without vendor hooks?

Qais Yousef

Nov 2023

Agenda

01 Introduction

02 Upstream feature limitations and bugs

03 Tunability

04 How can we break the deadlock?

05 Q&A

01
Introduction

What are vendor hooks?

- Vendor hooks are trace points that modules can attach to
- They either extend or override aspects of a kernel function
- Allows vendors to extend/modify Android Generic Kernel Image (GKI) without carrying or modifying these changes in Android kernel tree.
- They help vendors address problems that can’t be fixed easily in GKI
- The goal is to work with upstream to make these vendor hooks reduced over time

Can mainline Linux run on Android without vendor hooks?

Except for a small number of required patches to enable booting android; we certainly can. But at the expense of final user experience.

- UI will be jittery
- Reduced battery life
- Thermal throttling will be more prominent
- Potentially missing on hardware features

Performance and power management are major issues especially under the constrained conditions the mobile devices must operate under. Different devices are built for different trade-offs and goals that can’t be done on vanilla upstream kernel.

Scheduler is a major problem..

Scheduler is one the tricky area that continues to be heavily modified out-of-tree. Android Generic Kernel Image (GKI) has reduced the diff against upstream, especially after EAS got merged, but vendors still continue to modify it via vendor hooks.

The reasons can be broken down roughly into 2 major categories

- Feature limitations and bugs in upstream
- Tunability

Another indirect major problem is the extremely fast pace of mobile world where a new SoC is released every year. Compared to the slower development process upstream; especially scheduler subsystem as it has a very wide and noticeable impact on everything runs Linux, addressing these issues is very challenging in practice. For example it is becoming hard to use one Energy Model to represent the hardware for all possible workloads.

.. but not the only one

- General power and thermal management in mobile environment is very challenging. Mainline kernel is tuned for perf first.
- Due to fast hardware turn around, drivers being upstreamed is a very hard target
- Silicon erratas need to be handled long before they're available upstream or can't be upstreamed altogether.
- Android is a differentiating market; uniformity is desired to a limited extent for business reasons. Enabling selecting different trade-offs helps address this need.
- Userspace interface/libraries for managing big.LITTLE is still not up to par. This is not yet considered a community problem and vendors provide their own SDKs built around their own kernel interfaces. Kernel/userspace interfaces didn't evolve enough given progress in hardware and software in the past decades.
- Mainline kernel doesn't cater for context based system tuning in general. There's no one size fits all. We need more situational based tuning to be made available.
    - There's an inherent assumption the system is static with pre-knowledge what it'll do.

02

Upstream feature limitations and bugs

Fair is not good enough

- A lot of the scheduler problems try to fight SCHED_NORMAL fairness
- Not all tasks are equal and they need different latency and performance requirements that can’t be addressed completely yet
    - Uclamp enabled managing performance
    - Hopefully upcoming latency interface will allow to address the other problem
    - Yet to be seen, but has been many indication for the need for something that is inbetween RT and CFS

Lack of mechanism to annotate tasks' requirement or desired behavior

- We need a generic QoS framework to help classify the requirements of the tasks and their desired behavior
- Wake up latency is one prominent missing interface that is essential for today's Android operation. Mainly for UI, but not limited to it.
- There are packing vs spreading behavior type of workloads that can't be annotated today.
    - EAS has a packing behavior vs scheduler default spreading.
- GKI provides prefer_idle which provides better latency and avoids packing
    - But it is very expensive in terms of power and no per-task interface is available
    - Vendor hooks end up working around these limitations under various corner cases
    - GKI doesn't handle load balancing of latency sensitive tasks

Inversions and starvations problems

- cpu.shares can’t be used due to priority inversion and starvation problems
  - This makes the whole cpu cgroup controller unusable
  - Proxy execution has been brewing for years but not yet close enough to be used in products
- Performance inversion is another problem due to big.LITTLE and DVFS
  - Task A blocked on Task B that is running on little core/low frequency leading to big delays with noticeable impacts on performance

Background tasks can steal perf and power

- Currently cpusets are used to keep these tasks in check
  - But it is not a global win and introduces another class of problems
- uclamp_max can help but has issues due to perf first bias in current implementation that makes it not effective enough in practice
- Load balancer and wake up path don’t handle keeping these tasks out of other important tasks way
  - Bandwidth controller can help but has limitations that needs to be addressed
  - Bandwidth controller, shares etc introduces inversion and starvation problems

Energy awareness for RT and interoperability

- Task placement is randomly done based on lowest priority CPU
- With uclamp_min we can make them run at lower frequency, but not control which CPU that can lead to lowest power consumption
- Poses more difficult questions whether RT is lacking energy awareness or SCHED_NORMAL needs to evolve more so users don't need to use RT under these circumstances
- Latency sensitive SCHED_NORMAL and RT tasks must avoid being put on the same CPU
    - This type of inter-sched-class awareness doesn't exist today

Bugs take time to fix

- A vendor hook fix for a bug in upstream kernel takes a day or two
- Fixing it upstream takes months
  - Propagation time to LTS and products is an extra overhead on top
- Sometimes it is hard to know whether the bug is actually in upstream due to all interdependencies
  - Extra delay
  - Should be less of a problem when vendor hooks are reduced to bare minimum

03
Tunability

Design for tunability - allow userspace to select trade-offs

- Android is a general purpose system that needs to handle a wide range of scenarios and workloads
- There is no one size fits all solution
- A lot of effort was put into classifying situations to enable doing the right thing for this context by a smart middleware
    - Ie: screen is off enables taking advantage to limit frequencies
    - Ie: games can have different perf/watt trade-off compared to other apps
- Kernel doesn't have enough mechanism to enable context based tuning
    - Boot time tuning is reasonable for some elements, but not all
    - DVFS response time is one example of such runtime tunable
- **This is NOT about introducing sched tunables!**
- But there are trade-offs to be made. We need to give userspace some power to control these trade-offs in a sensible and meaningful way.

04

How can we break the deadlock?

Should we delegate more to userspace?

- We need a paradigm shift. We can’t fix everything in the kernel.
- Traditionally improving the scheduler default behavior was the way to go.
- We must move away from this mindset and think more how apps can be written to take better advantage of the hardware in a portable way.
- Android userspace won’t move to using non standard kernel interfaces.
- App developers shouldn’t see programming for Android is different than programming for Linux in general (Desktop or Server).

Is our programming model still adequate?

- We have moved a long way from systems being limited by perf and elements like power and thermal are increasingly more limiting factor.
    - Even in the server markets that are traditionally associated with Linux.
- Workloads that are expected to run on any system, including servers, has increased that make the job of a team responsible for tuning it harder as the number of conflicting requirements has increased as well.
- Programming interface for Linux which is mainly defined by POSIX has stalled and didn't keep up with modern changes in hardware/software.
    - Don't quote me, but other OSes provide richer non-POSIX interfaces.
- We need to move away from the old method of improving default behavior and give userspace more responsibility to earn their best perf/watt by providing mechanisms and avoid plugging accidental policies in the kernel to improve the 'default behavior'.
- Portability of the applications that Just Work on any system is essential.

Who owns trade-offs responsibility and how they should be made?

- There are trade-offs to be made. And userspace need to have a say in making these trade-offs.
- The prominent use case of the system can change over time or per user.
- Each use case might require its own trade-offs that can’t be achieved in a generic way.
- Distilling this into a set of generic tunables or programming interfaces is necessary to enable extracting the most out of the device to achieve the best outcome for a specified task/service.
- Again, **this is NOT about tuning scheduler implementation details**. But a higher level control of system behavior.
- Both kernel and userspace need to communicate their needs and limitations and then allow each to do the right thing ™.

What’s the right collaboration model for such changes?

- There’s a chicken and egg problem
- We can’t deploy out of tree without upstreaming first, but upstream won’t accept without enough data to prove it indeed solves the problems adequately
- Early feedback from community is important
- Full solution and clarity on requirement is hard to present early
- Full testing coverage sometimes requires some sort of deployment out of tree first
- Is there a recommended way to handle this type of problems?

05
Q&A