Google

There’s a black hole in the scheduler

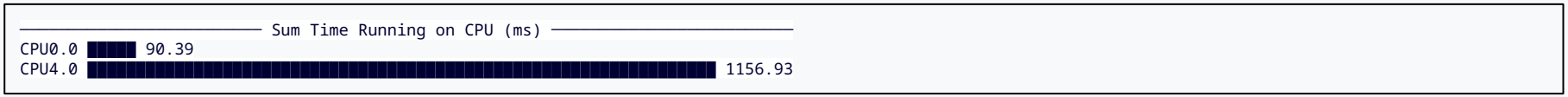
Better management of system response time

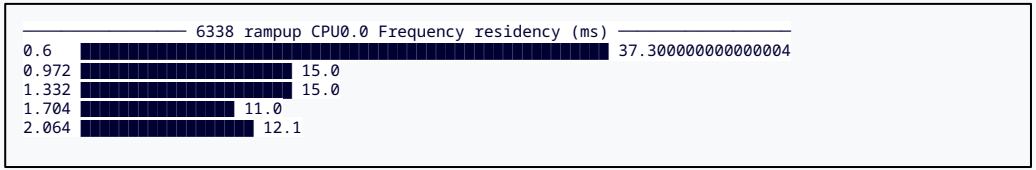
Qais Yousef

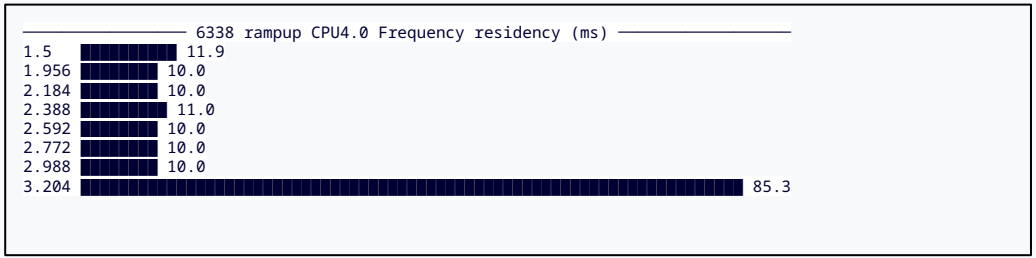
Sep 2024

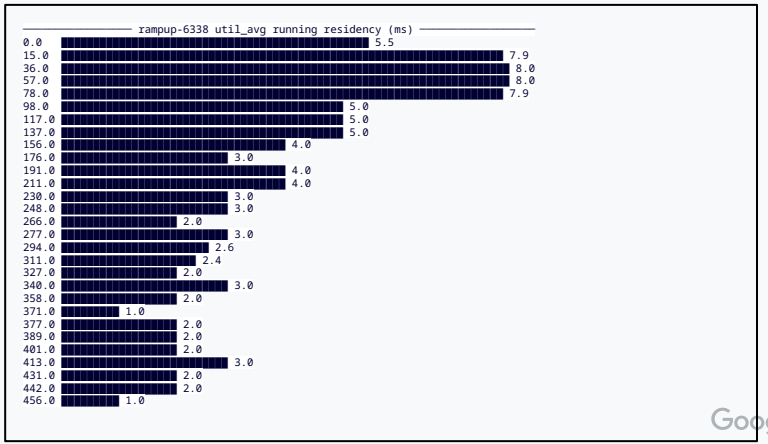
DVFS and HMP slow down time

- The lower the frequency/capacity, the longer it takes to finish the same amount of work
- Utilization invariance introduces Black Hole effect of Time Dilation
  - 1ms for a task is actually 25-30ms in real time (from observer's/userspace PoV)
- Always busy task starting from util_avg/util_est = 0 appears as stuck on little core for 90ms, 37ms of which running at lowest frequency



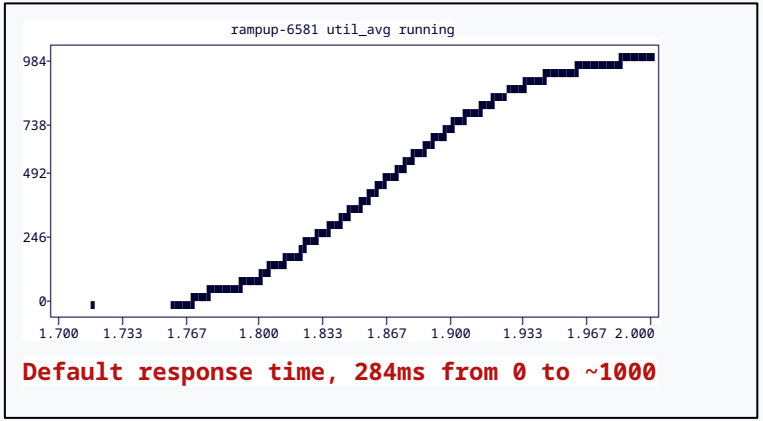




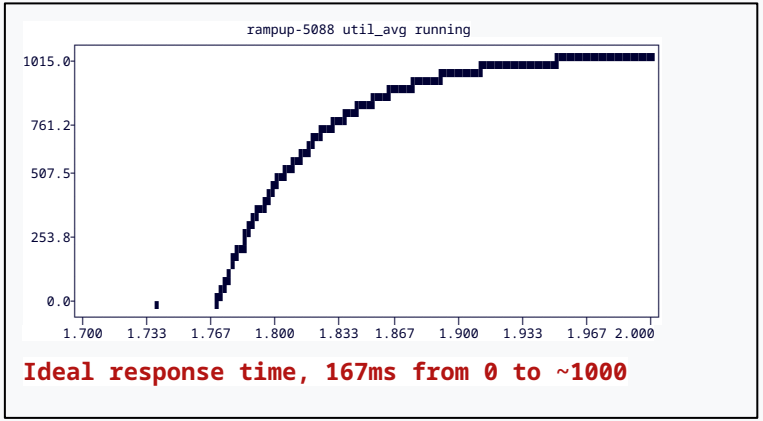


Ideal response time is running at perf level = 1024

- User space expects tasks utilization to uniformly rampup as if they are running with a performance governor all the time
- Under this circumstances where util invariance is effectively a NOP, the utilization value grows every tick (1ms in this example)



Default response time, 284ms from 0 to ~1000



Ideal response time, 167ms from 0 to ~1000

Impacts on scheduler decisions

- Schedutil governor will appear unresponsive on many systems
    - Long TICK and rate_limit_us compound this this problem
- On HMP systems tasks appear being ‘stuck’ on underperforming cores after prolonged period of activity
- System will appear less loaded for prolonged period of time, leading to ineffective load balancing and more wrong decisions at wakeup path
- Migration margins and DVFS headroom are currently hardcoded based on old system properties that I think were suitable to hide this effect then, but they need to change now as they cause either bad perf or bad power on different type of systems/workloads

Impact on fairness

- Do we manage vruntime adequately to reflect this time dilation impact?
  - If a task is running for 1ms on little core moves to a big core that has a task already running for 1ms, who is more viable for CPU time so that they both had access to the same computational demand?
  - Do we need debt_vruntime concept?
- If we define waiting_avg as the time in RUNNABLE && !RUNNING
  - If a task had a waiting_avg of 6ms on little core and moves to a bigger core that already has 2 tasks running, who has preference to preempt next so that their waiting_avg is equivalent?
- An example of such fairness problem is always running tasks on N cores HMP systems or a system with multiple independent cpufreq policies. Many implement task rotation mechanism as this is more fair. Addressing the above would be a better way to fix the problem.

What can we do about it?

- Extend util_est to behave like ideal response when tasks are transient
  - Perfectly periodic tasks have no problem by definition and current util_est behavior is sufficient
  - Periodic tasks are tasks that have their util_avg the same across activations
  - Transient tasks are ones that have their util_avg rising across activations
- Remove hardcoded migration margin and DVFS headroom with more automatic one based on worst case scenario
- rampup_multiplier to give userspace the choice to go faster or 0 to indicate slow rampup is okay which can help to save power
- Introduce waiting_avg to improve DVFS headroom and to potentially better handle latencies in load_balance/wakeup path
- No idea about vruntime (if it is indeed a problem)

Questions