Schedutil

Note

All this assumes a linear relation between frequency and work capacity, we know this is flawed, but it is the best workable approximation.

PELT (Per Entity Load Tracking)

With PELT we track some metrics across the various scheduler entities, from individual tasks to task-group slices to CPU runqueues. As the basis for this we use an Exponentially Weighted Moving Average (EWMA), each period (1024us) is decayed such that y^32 = 0.5. That is, the most recent 32ms contribute half, while the rest of history contribute the other half.

Specifically:

> ewma_sum(u) := u_0 + u_1*y + u_2*y^2 + ...
>
> ewma(u) = ewma_sum(u) / ewma_sum(1)

Since this is essentially a progression of an infinite geometric series, the results are composable, that is ewma(A) + ewma(B) = ewma(A+B). This property is key, since it gives the ability to recompose the averages when tasks move around.

Note that blocked tasks still contribute to the aggregates (task-group slices and CPU runqueues), which reflects their expected contribution when they resume running.

Using this we track 2 key metrics: 'running' and 'runnable'. 'Running' reflects the time an entity spends on the CPU, while 'runnable' reflects the time an entity spends on the runqueue. When there is only a single task these two metrics are the same, but once there is contention for the CPU 'running' will decrease to reflect the fraction of time each task spends on the CPU while 'runnable' will increase to reflect the amount of contention.

For more detail see: kernel/sched/pelt.c

Frequency / CPU Invariance

Because consuming the CPU for 50% at 1GHz is not the same as consuming the CPU for 50% at 2GHz, nor is running 50% on a LITTLE CPU the same as running 50% on a big CPU, we allow architectures to scale the time delta with two ratios, one Dynamic Voltage and Frequency Scaling (DVFS) ratio and one microarch ratio.

For simple DVFS architectures (where software is in full control) we trivially compute the ratio as:

```
          f_cur
r_dvfs := -----
          f_max
```

For more dynamic systems where the hardware is in control of DVFS we use hardware counters (Intel APERF/MPERF, ARMv8.4-AMU) to provide us this ratio. For Intel specifically, we use:

```
         APERF
f_cur := ----- * P0
         MPERF

           4C-turbo;  if available and turbo enabled
f_max := { 1C-turbo;  if turbo enabled
           P0;        otherwise

                  f_cur
r_dvfs := min( 1, ----- )
                  f_max
```

We pick 4C turbo over 1C turbo to make it slightly more sustainable.

r_cpu is determined as the ratio of highest performance level of the current CPU vs the highest performance level of any other CPU in the system.

> r_tot = r_dvfs * r_cpu

The result is that the above 'running' and 'runnable' metrics become invariant of DVFS and CPU type. IOW. we can transfer and compare them between CPUs.

For more detail see:

- kernel/sched/pelt.h:update_rq_clock_pelt()
- arch/x86/kernel/smpboot.c:"APERF/MPERF frequency ratio computation."

- Documentation/scheduler/sched-capacity.rst:"1. CPU Capacity + 2. Task utilization"

UTIL_EST / UTIL_EST_FASTUP

Because periodic tasks have their averages decayed while they sleep, even though when running their expected utilization will be the same, they suffer a (DVFS) ramp-up after they are running again.

To alleviate this (a default enabled option) UTIL_EST drives an Infinite Impulse Response (IIR) EWMA with the 'running' value on dequeue -- when it is highest. A further default enabled option UTIL_EST_FASTUP modifies the IIR filter to instantly increase and only decay on decrease.

A further runqueue wide sum (of runnable tasks) is maintained of:

> util_est := Sum_t max(t_running, t_util_est_ewma)

For more detail see: kernel/sched/fair.c:util_est_dequeue()

UCLAMP

It is possible to set effective u_min and u_max clamps on each CFS or RT task; the runqueue keeps an max aggregate of these clamps for all running tasks.

For more detail see: include/uapi/linux/sched/types.h

Schedutil / DVFS

Every time the scheduler load tracking is updated (task wakeup, task migration, time progression) we call out to schedutil to update the hardware DVFS state.

The basis is the CPU runqueue's 'running' metric, which per the above it is the frequency invariant utilization estimate of the CPU. From this we compute a desired frequency like:

```
              max( running, util_est );  if UTIL_EST
   u_cfs := { running;                   otherwise

                clamp( u_cfs + u_rt , u_min, u_max );    if UCLAMP_TASK
   u_clamp := { u_cfs + u_rt;                            otherwise

   u := u_clamp + u_irq + u_dl;          [approx. see source for more detail]

   f_des := min( f_max, 1.25 u * f_max )
```

XXX IO-wait: when the update is due to a task wakeup from IO-completion we boost 'u' above.

This frequency is then used to select a P-state/OPP or directly munged into a CPPC style request to the hardware.

XXX: deadline tasks (Sporadic Task Model) allows us to calculate a hard f_min required to satisfy the workload.

Because these callbacks are directly from the scheduler, the DVFS hardware interaction should be 'fast' and non-blocking. Schedutil supports rate-limiting DVFS requests for when hardware interaction is slow and expensive, this reduces effectiveness.

For more information see: kernel/sched/cpufreq_schedutil.c

NOTES

- On low-load scenarios, where DVFS is most relevant, the 'running' numbers will closely reflect utilization.
- In saturated scenarios task movement will cause some transient dips, suppose we have a CPU saturated with 4 tasks, then when we migrate a task to an idle CPU, the old CPU will have a 'running' value of 0.75 while the new CPU will gain 0.25. This is inevitable and time progression will correct this. XXX do we still guarantee f_max due to no idle-time?
- Much of the above is about avoiding DVFS dips, and independent DVFS domains having to re-learn / ramp-up when load shifts.