The "schedutil" frequency scaling governor

Giovanni Gherdovich
ggherdovich@suse.cz

October 7th, 2018

Agenda

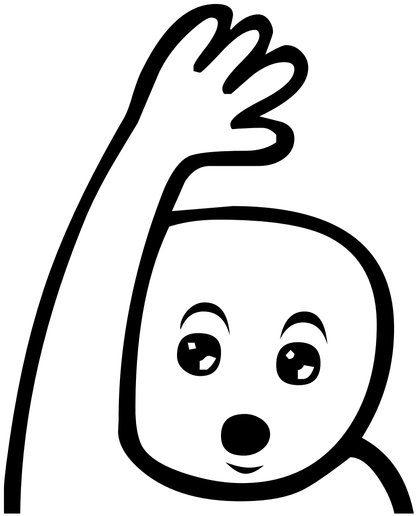
> schedutil intro

> frequency scale invariance

> PELT

> util_est

Questions, anytime



terminology

> freq scaling governor:
    > algorithm ("policy") to decide which freq to run next
    > eg: ondemand, powersave (intel specific), schedutil, ...
> freq scaling driver:
    > communicates to the hardware the desired setting
    > eg: acpi_cpufreq, pcc_cpufreq, intel_pstate, intel_cpufreq, ...

terminology

What am I running?

```
$ cpupower frequency-info --driver
$ cpupower frequency-info --policy
$ cpupower frequency-info --governors
```

Agenda

> schedutil intro

> frequency scale invariance

> PELT

> util_est

schedutil

> generic frequency governor (works with multiple drivers)
> works from scheduler data (PELT utilization signal)
> utilization signal is per-task (migrates with task_struct)
> merged in v4.7 (April 2016)

> compare with intel_pstate/powersave: CPU utilization data from APERF / MPERF registers

Agenda

> schedutil intro

> frequency scale invariance

> PELT

> util_est

frequency scale invariance

Tasks appear larger if CPU is running slower.

⇨ dividing current freq by max freq gives invariant utilization metric

frequency scale invariance

> Utilization, Load: arbitrary cost functions

> dimensionless quantities

> utilization should be between 0 (empty) and 1 (full)

> we want to define them per-task

dumb example: utilization of a task is the percentage of running time (`se->on_cpu`) during last millisecond.

⇨ lower if CPU runs faster

⇨ ill-defined, meaningless

frequency scale invariance

solution: multiply dumb utilization by $\texttt{freq}_{curr}$ / $\texttt{freq}_{max}$

> still dumb, but scale invariant!
> merged for ARM in v4.15 (January 2018)

new problem: x86 doesn't have $\texttt{freq}_{max}$, turbo states availability depends on neighboring cores

> patch floating around, dynamic discovery of $\texttt{freq}_{max}$ reading the APERF and MPERF registers

frequency scale invariance

- > schedutil formula
  - > utilization is frequency invariant (ARM):

$$freq_{next} = 1.25 * freq_{max} * util$$

  - > utilization is not frequency invariant (x86):

$$freq_{next} = 1.25 * freq_{curr} * util$$

frequency scale invariance

- > schedutil formula
  - > utilization is frequency invariant (ARM):

$$freq_{next} = 1.25 * freq_{max} * util$$

frequency scale invariance

> schedutil formula

  > utilization is frequency invariant (ARM):

$$freq_{next} = 1.25 * freq_{max} * util$$

    > rationale: make $freq_{next}$ proportional to util

    > since `1.25 * 0.8` is `1`, when util is `0.8` sets freq to max

    > we consider 80% a high utilization, so better speed up

    > note: after switching freq, utilization remains the same

frequency scale invariance

> schedutil formula

> > utilization is not frequency invariant (x86):

$$freq_{next} = 1.25 * freq_{curr} * util$$

frequency scale invariance

> schedutil formula

  > utilization is not frequency invariant (x86):

$$freq_{next} = 1.25 * freq_{curr} * util$$

    > derived from the invariant case, replace

$$util_{inv} = util_{raw} * freq_{curr} / freq_{max}$$

    > approximation: $util_{raw}$ is a PELT sum, each term needs to be scaled (with $freq_{curr}$ at that time)

    > $util_{raw}$ == 0.8 is the tipping point: less than 0.8 and freq goes down, more than 0.8 and freq goes up

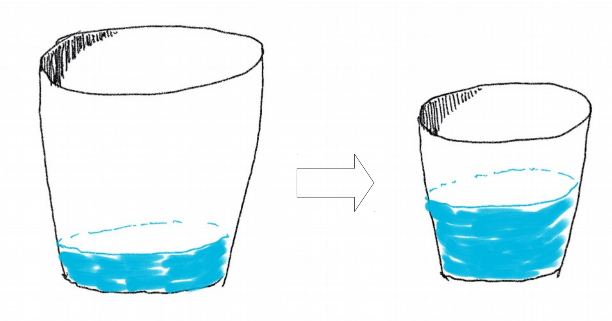
frequency scale invariance

> metaphore for the non invariant case: bucket of water

You’re given a bucket F with some water W. Let’s call U the ratio of water volume by the total:

$$U = W / F$$

Find the volume of a new bucket F’ to pour the water into so that the new utilization U’ = W / F’ is 0.8.



frequency scale invariance

> metaphore for the non invariant case: bucket of water

You’re given a bucket F with some water W. Let’s call U the ratio of water volume by the total:

$$U = W / F$$

Find the volume of a new bucket F’ to pour the water into so that the new utilization U’ = W / F’ is 0.8.

$$0.8 = W / F'$$

$$\Rightarrow F' = 1.25 * W$$

$$\Rightarrow F' = 1.25 * F * U$$

frequency scale invariance

> metaphore for the non invariant case: bucket of water

> > water bucket: F is total volume, W is water volume
> > freq switching: F is current frequency, W is instructions per second (“useful work”).

> > if F is cycles per second, U = W / F would give instruction per cycle (IPC). Maybe?

frequency scale invariance

- > schedutil formula
  - > utilization is frequency invariant (ARM):

$$freq_{next} = 1.25 * freq_{max} * util$$

  - > utilization is not frequency invariant (x86):

$$freq_{next} = 1.25 * freq_{curr} * util$$

Agenda

> schedutil intro

> frequency scale invariance

> **PELT**

> util_est

Per Entity Load Tracking (PELT, v3.8, Oct. 2012)

> “PELT” is a property of `struct sched_entity`

> recursively defined:

  > “PELT” on groups and runqueues is the sum of “the PELT’s” of their constituents

  > “PELT” on tasks is the sum of past runnable (load) or running (util) times (see next slides)

> “PELT” is actually two numbers:

  > `load_avg`, used by for eg. load balancing

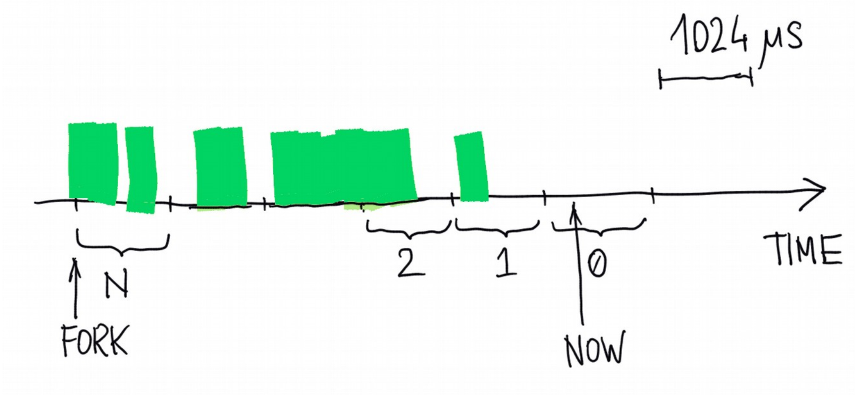
  > `util_avg`, used for eg. in schedutil

    > almost identical formula, but runnable time replaced by running time

Per Entity Load Tracking (PELT)

- > load_avg and util_avg are our cost functions
  - > partition time into segments of 1024 µs
  - > segments aligned with task creation

$$util = \frac{R_0 + R_1 y + R_2 y^2 + R_3 y^3 + \ldots + R_N y^N}{1024\left(1 + y + y^2 + y^3 + \ldots + y^N\right)}$$



- > y = 0.9785
- > R_i is time (µs) in segment i …
  - > util_avg: … the task was running
  - > load_avg: … the task was runnable
- > dimensionless
- > $util_{new} = util_{old} * y + R_0$

Per Entity Load Tracking (PELT)

```
#!/bin/bash
for i in {1..10} ; do
        N=0
        while true ; do
                ((N++))
        done &
done
```

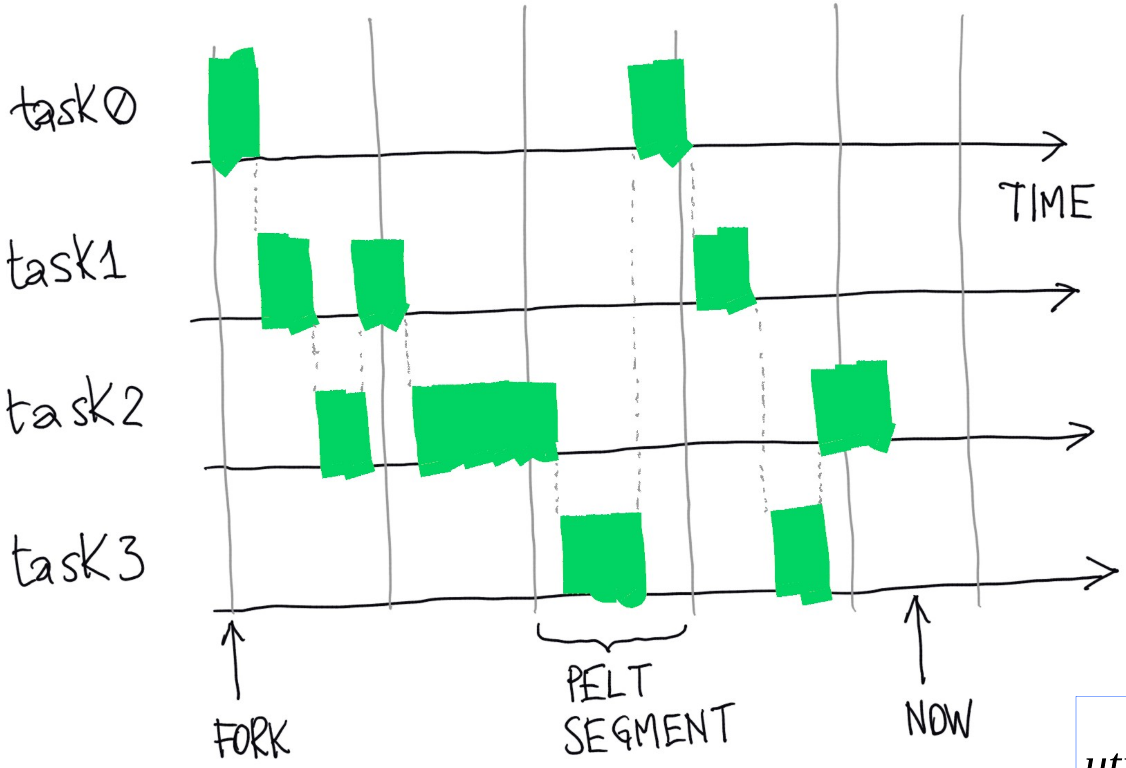
```
$ taskset --cpu-list 0 ./heavy.sh
$ echo t > /proc/sysrq-trigger
```

Per Entity Load Tracking (PELT)

```
cfs_rq[0]:/
...
.nr_running                    : 10
...
.load_avg                      : 10239
.runnable_load_avg             : 10239
.util_avg                      : 1023
.util_est_enqueued             : 10
...
```

> toplevel runqueue for cpu#0
> 1024 is 1 in fixed point arith
> load_avg unbound
> util_avg bound by 1024…
> > why?

Per Entity Load Tracking (PELT)



$$util = \frac{R_0 + R_1 y + R_2 y^2 + R_3 y^3 + \ldots + R_N y^N}{1024\left(1 + y + y^2 + y^3 + \ldots + y^N\right)}$$

Per Entity Load Tracking (PELT)

```
# cat /proc/$$/sched
bash (15127, #threads: 1)
-------------------------------------------------------------------
...
se.avg.load_sum                          :                  629
se.avg.runnable_load_sum                 :                  629
se.avg.util_sum                          :               620282
se.avg.load_avg                          :                    0
se.avg.runnable_load_avg                 :                    0
se.avg.util_avg                          :                    0
se.avg.last_update_time                  :      199010878882816
se.avg.util_est.ewma                     :                    8
se.avg.util_est.enqueued                 :                    0
...
```

peek at a process' PELT data

Agenda

> schedutil intro

> frequency scale invariance

> PELT

> util_est

util_est, improved responsiveness

> signal built on top of PELT
> computed only for tasks and top level runqueues
> stores util_avg at dequeue, before it decays
> merged in v4.17 (March 2018)
> schedutil now consumes max(util_est, util_avg)

util_est, improved responsiveness

util_est is a struct of two numeric fields:

> enqueued:
  > for a task: util_avg at the time of last dequeueing
  > for a cfs_rq: for each task take max(enqueued, ewma) and sum
> ewma:
  > for a task: Exponentially Weighted Moving Average of past util_avg’s at dequeue
    > keeps memory of last few dequeues, “ignores” false restarts

Thanks!

> PELT (2012) introduces per-task util tracking in the scheduler
> schedutil (2016) uses PELT data to drive freq scaling
> util_est “caches” util data from previous dequeues to make PELT ramp up faster
  > and considerably improves schedutil
> schedutil re-claims a privileged position for the OS in freq scaling
  > the hardware is oblivious of tasks, migrations, etc
> schedutil requires freq-invariant utilization

Extras

util_est, improved responsiveness

$$\texttt{ewma}_t = 0.25 * \texttt{util\_avg}_t + 0.75 * \texttt{ewma}_{t-1}$$

eliminating recursion:

$$\texttt{ewma}_{now} = 0.25 * \Sigma_k\{0.75^k * \texttt{util\_avg}_k\}$$

re-labeled terms so that:

k = 0 is last dequeueing,

k = 1 is penultimate dequeuing,

k = 2 is two dequeueings before the last, etc

`0.75 ^ 2.409 = 0.5`

⇨ half life of weight is between
2 and 3 dequeuing (memory span)

P-States facts (x86)

Arjan van de Ven ▸ Public   Jun 23, 2013

Some basics on CPU P states on Intel processors

there seems to be a lot of things people don't realize on how P state selection works on Intel processors, and arguably the documentation is slightly confusing in this regard... and things have been changing generation to generation.

...

P-States facts (x86)

> all cores in a package share same voltage V

> running at lower freq than possible (given V) is inefficient

⇨ all cores (non idle) share the same clock freq F (!?!)

⇨ F is the max requested by OS for any of the (non idle) cores

statistically significant | meh | statistically significant

benchmarks: vanilla 4.17

intel_pstate/powersave VS intel_cpufreq/schedutil

MMTESTS CONFIG		2 x BROADWELL 80 CORES	1 x SKYLAKE 8 CORES	UNIT	BETTER IF
db-pgbench-timed-ro-small	pgbench	1	1.01	TRANS_PER_SECOND	higher
io-dbench4-async	dbench4	1.06	1	TIME_MSECONDS	lower
network-netperf-unbound	netperf-tcp	1.02	1	MBITS_PER_SECOND	higher
	netperf-udp	0.99	0.99	MBITS_PER_SECOND	higher
network-sockperf-unbound	sockperf-tcp-throughput	1.97	0.99	MBITS_PER_SECOND	higher
	sockperf-tcp-under-load	1.02	0.96	TIME_USECONDS	lower
	sockperf-udp-throughput	1	0.99	MBITS_PER_SECOND	higher
	sockperf-tcp-under-load	0.83	0.97	TIME_USECONDS	lower
scheduler-unbound	hackbench-process-pipes	0.99	0.99	TIME_SECONDS	lower
	hackbench-process-sockets	0.99	0.99	TIME_SECONDS	lower
	hackbench-thread-pipes	1.01	1.02	TIME_SECONDS	lower
	hackbench-thread-sockets	0.96	0.99	TIME_SECONDS	lower
	pipetest	1.89	2	TIME_USECONDS	lower
workload-kerndevel	gitcheckout	1.03	1.02	TIME_SECONDS	lower
	kernbench	1.08	1.04	TIME_SECONDS	lower
workload-schbench	schbench	1.09	1.07	TIME_USECONDS	lower
workload-shellscript	gitsource	1.02	1.39	TIME_SECONDS	lower