Android Scheduling

Adina Johnson

1 Introduction

Energy efficiency is the biggest driver for changes in Android scheduling, as lower energy consumption is much more of an issue in the small, battery-powered mobile devices using Android than the bigger desktops and laptops running Linux (although some of these changes are starting to come to Linux as laptop battery life becomes more of a concern). Android provides two new frameworks, WALT and SchedTune, described below. In /kernel/sched/, Android provides four new files: walt.c, tune.c, energy.c, and cpufreq_sched.c. In addition, there are differences in the files (between Android Common 4.9 and Linux 4.9) fair.c, core.c, rt.c, sched.h, cputime.c, debug.c, features.h, idle.c, and stop_task.c. This document first provides an overview of the changes, then documentation of functions found in Linux that are modified in the Android kernel, as well as new functions introduced by Android.

2 Glossary

The kernel is complicated and full of new terms and acronyms. This glossary attempts to clarify some scheduling-relevant terms.

sd: scheduling domain
"A set of CPUs which share properties and scheduling policies, and which can be balanced against each other." See Scheduling Domains for more information.

rq: run queue
A data structure to hold runnable processes on a given processor. See The Linux Process Scheduler for more information.

rt: real time
Real-time scheduling is explained well in this article. In the kernel, it is an alternative to the standard scheduling algorithm and is optimal for time-critical applications that need more precise control.

nr: number
This is found in some functions in sched.h.

big.LITTLE:
def

cfs: Completely Fair Scheduler
This is the standard Linux scheduling algorithm. See Inside the Linux 2.6 Completely Fair Scheduler for more information.

dt: device tree

3 Overview of New Systems

3.1 Energy-Aware Scheduling (EAS)

Energy-Aware Scheduling encompasses most of the changes added to Android. Linaro, the main developer of the system, has a good overview in their blog post Energy Aware Scheduling (EAS) progress update. EAS tweaks scheduling algorithms in general, as well as linking the infrastructure for CPU power management with the scheduler, in an effort to schedule tasks in a more energy-efficient manner without affecting performance. One change is to the task assignment algorithm: in Linux, the "Completely Fair Scheduler" (CFS), the main scheduling algorithm, will always put a new task on an idle CPU if available. However, this is not always the most energy-efficient decision. EAS also attempts to make the scheduler more compatible with big.LITTLE systems, thus changing the assumption in the scheduler that all CPUs are equal.

Nicolas Pitre's article Teaching the scheduler about power management provides a good overview of the two CPU power-management systems linked to the scheduler by EAS. One of these systems, the CPUIdle subsystem attempts to minimize power consumption by selecting a low-power or idle mode when the CPU has no tasks running. However, the more power savings a mode gives, the longer it will take to wake the CPU up. Additionally, many modes consume energy when shifting in and out of them, meaning the CPU should be idle for a sufficient period of time for entering the mode to be worth it. Most CPUs have multiple idle modes with different tradeoffs between power savings and latency. Thus, the CPUIdle code needs to gather data on CPU usage to select an appropriate mode. This duplicates work the scheduler does (albeit imperfectly). Linking the scheduler with the CPUIdle subsystem, systems which previously operated separately from each other, gives the scheduler awareness of the current idle mode on each CPU and allows the scheduler to pick the CPU in the shallowest idle-state when waking one up, minimizing wake-up time and energy.

The CPUFreq subsystem is the second system linked to the scheduler. CPU frequency is the frequency of CPU's core. The higher the frequency, the faster the processor, but this also means the processor is consuming more energy. *cpufreq* in the kernel allows for frequency scaling (modifying the CPU frequency). This system also duplicates work done by the scheduler and thus is integrated in EAS.

In general, when picking a CPU for a task to run on, the EAS policy is to find a CPU with enough capacity for the task that has the smallest energy impact. Because evaluating all possible options would impact performance, EAS simply evaluates the CPU the task last ran on and the CPU chosen by a simple heuristic which finds where the task best fits. EAS then evaluates which option is the most energy efficient. Some tasks prefer idle CPUs, which the scheduler will take into consideration. Juri Lelli's slide presentation provides a good overview of this algorithm.

EAS also uses WALT and SchedTune, two major additions to the kernel described in more detail below.

3.2 WALT Framework

The WALT (Window-Assisted Load Tracking) framework is a modification of the PELT (Per-Entity Load Tracking) framework. Both are meant to measure the "load" of a task. The load is essentially how much of the CPU the task is using, although Jonathan Corbet notes in his article "Per-entity load tracking" that there is a difference between "CPU time consumed" and "load." Namely, a process waiting its turn to run can still contribute to load, and load is an instantaneous quality; for example, a task whose past behavior was very demanding may contribute very little to load at the present if it currently places modest demands on the CPU.

Window-Assisted Load Tracking (WALT) is introduced as an alternative or additional load tracking scheme, improving upon Per-Entity Load Tracking (PELT) for mobile devices in order to better track task demand and CPU utilization. WALT has performed better than PELT in limited device testing. The commit message for the WALT patch by Vikram Mulukutla compares WALT and PELT: WALT "use[s] a window based view of time in order to track task demand and CPU utilization in the scheduler," and both account for demand and utilization at the task level and consider a CPU runqueue's utilization contribution the sum of its children's contributions. WALT, however, uses different time measurements.

In WALT, wallclock time is divided into a series of windows. Task demand is the contribution of a task's running and runnable time to a window, averaged over N sliding windows. The sliding windows exist only when the task is on the runqueue or running (i.e. it "forgets" blocked time). It also enables the system to classify a task as "heavy" faster, which is important on mobile phones where tasks experience sporadically heavy load, such as web browsing. This allows for faster migration to a better-suited CPU. Mulukutla writes that this requires "a more dynamic demand or utilization signal than provided by PELT, which is subject to the geometric series and does not differentiate between task ramp-up and ramp-down." WALT also tracks all tasks with a single signal, while PELT only tracks utilization of SCHED_OTHER classes.

WALT-related functions are defined in walt.c and used in core.c, cputime.c, fair.c, rt.c, sched.h, and stop_task.c.

3.3 SchedTune

SchedTune is implemented as a control-group controller, described in the article Scheduling for Android devices. The kernel needs to be able to switch between prioritizing fast performance and energy conservation in an easy and efficient way. SchedTune, built on top of PELT signals and sched-DVFS, is able to tune the operating frequency to better match the workload demand. It does this by signal boosting, or artificially inflating some load tracking signals, as described in the Sched-Tune Documentation. SchedTune-related functions are defined in tune.c and fair.c

4 Analysis of Files

4.1 energy.c

This file extracts energy cost data from DT and fills in relevant scheduler data structures. It only has two functions, and one is simply a helper function to free resources after the main function, init_sched_energy_costs, is complete.

4.2 fair.c

Many of the new functions in fair.c are related to SchedTune. update_capacity_of is called multiple times throughout the file. This function is related to SchedTune and it just changes CFS CPU capacity (updating the OPP of the target).

cpu_overutilized is a new concept in Android. Energy-aware scheduling should only be active when the system is not over-utilized–when there are spare cycles available to move tasks around based on their actual utilization to get a more energy-efficient task distribution without depriving any tasks. Additionally, a energy-aware system that is not overutilized should be considered balanced. See the article Add over-utilization/tipping point indicator.

In need_active_balance: in Linux the last possible situation before an unlikely default return is one in which dst_cpu is idle and the src_cpu CPU has only 1 CFS task. The function checks if the src_cpu's capacity is reduced because of other sched_class or IRQs, in which case it is worth migrating the task (if more capacity stays available on dst_cpu). In Android, there is an additional conditional check before the default return: another check for a situation in which the CPU capacity of the destination is greater than that of the source, the source only has 1 running process but it is overutilized and the destination CPU isn't overutilized. In that case, we would migrate the task. See the code below:

```
if ((capacity_of(env->src_cpu) < capacity_of(env->dst_cpu)) &&
        env->src_rq->cfs.h_nr_running == 1 &&
        cpu_overutilized(env->src_cpu) &&
        !cpu_overutilized(env->dst_cpu)) {
    return 1;
}
```

select_idle_sibling has been changed to take C-State into account when selecting idle core. As described in the commit message, “When ‘sched_cstate_aware’ is enabled, select_idle_sibling in CFS is modified to choose the idle CPU in the sibling group which has the lowest idle state index - idle state indexes are assumed to increase as sleep depth and hence wakeup latency increase. This tries to minimise wakeup latency when an idle CPU is required.”

4.3 core.c

The majority of these changes are to add WALT support. Other changes are relevant to scheduling domains. Some of the new functions are energy-related, such as check_sched_energy_data and init_sched_energy. A patch by Sonny Rao notes that “Energy-aware scheduling allows that a system has only EM [Energy Model] data up to a certain sd level (so called highest energy aware balancing sd level). A check in init_sched_energy() enforces that all sd’s below this sd level contain EM data.” Thus, this file is also modified to make sure energy information is present and correct.

4.4 rt.c

Android has a new function dump_throttled_rt_tasks which, according to the commit, prints RT tasks when RT throttling is activated for debugging.

4.5 sched.h

There is a new struct added, max_cpu_capacity. Variables are added to structs defined in this file, such as a boolean for overutilization in struct root_domain, and a number of variables initialized for WALT. New functions relate to capacity and the number of running tasks.

5 Line-by-Line File Differences

5.1 fair.c

5.1.1 enqueue_task_fair

```
#ifdef CONFIG_SMP
    int task_new = flags & ENQUEUE_WAKEUP_NEW;
    int task_wakeup = flags & ENQUEUE_WAKEUP;
#endif
```

* * *

```
walt_inc_cfs_cumulative_runnable_avg(cfs_rq, p);
```

* * *

```
walt_inc_cfs_cumulative_runnable_avg(cfs_rq, p);
```

* * *

```
#ifdef CONFIG_SMP

    if (!se) {
            walt_inc_cumulative_runnable_avg(rq, p);
            if (!task_new && !rq->rd->overutilized &&
                cpu_overutilized(rq->cpu)) {
                    rq->rd->overutilized = true;
                    trace_sched_overutilized(true);
            }

            /*
             * We want to potentially trigger a freq switch
             * request only for tasks that are waking up; this is
             * because we get here also during load balancing, but
             * in these cases it seems wise to trigger as single
             * request after load balancing is done.
             */
            if (task_new || task_wakeup)
                    update_capacity_of(cpu_of(rq));
    }

    /* Update SchedTune accouting */
    schedtune_enqueue_task(p, cpu_of(rq));

#endif /* CONFIG_SMP */
```

5.1.2 dequeue_task_fair

```
walt_inc_cfs_cumulative_runnable_avg(cfs_rq, p);
```

* * *

```
walt_inc_cfs_cumulative_runnable_avg(cfs_rq, p);
```

* * *

```
#ifdef CONFIG_SMP

    if (!se) {
            walt_dec_cumulative_runnable_avg(rq, p);

            /*
             * We want to potentially trigger a freq switch
```

```
                 * request only for tasks that are going to sleep;
                 * this is because we get here also during load
                 * balancing, but in these cases it seems wise to
                 * trigger as single request after load balancing is
                 * done.
                 */
                                 if (task_sleep) {
                        if (rq->cfs.nr_running)
                                update_capacity_of(cpu_of(rq));
                        else if (sched_freq())
                                set_cfs_cpu_capacity(cpu_of(rq), false, 0);
                }
        }

        /* Update SchedTune accouting */
        schedtune_dequeue_task(p, cpu_of(rq));

#endif /* CONFIG_SMP */
```

5.1.3 find_idlest_group

```
struct sched_group *fit_group = NULL, *spare_group = NULL;
```

* * *

```
unsigned long fit_capacity = ULONG_MAX;
```

* * *

```
unsigned long max_spare_capacity = capacity_margin - SCHED_CAPACITY_SCALE;
```

* * *

```
/*
 * Look for most energy-efficient group that can fit
 * that can fit the task.
 */
if (capacity_of(i) < fit_capacity && task_fits_spare(p, i)) {
        fit_capacity = capacity_of(i);
        fit_group = group;
}

/*
 * Look for group which has most spare capacity on a
 * single cpu.
 */
spare_capacity = capacity_of(i) - cpu_util(i);
```

```c
if (spare_capacity > max_spare_capacity) {
        max_spare_capacity = spare_capacity;
        spare_group = group;
}
```

* * *

```c
if (fit_group)
        return fit_group;

if (spare_group)
        return spare_group;
```

5.1.4 find_idlest_cpu

```c
struct sched_group *fit_group = NULL, *spare_group = NULL;
```

* * *

```c
unsigned long fit_capacity = ULONG_MAX;
```

* * *

```c
unsigned long max_spare_capacity = capacity_margin - SCHED_CAPACITY_SCALE;
```

* * *

```c
#+BEGIN_SRC C
 /*
 * Look for most energy-efficient group that can fit
 * that can fit the task.
 */
if (capacity_of(i) < fit_capacity && task_fits_spare(p, i)) {
        fit_capacity = capacity_of(i);
        fit_group = group;
}

/*
 * Look for group which has most spare capacity on a
 * single cpu.
 */
 spare_capacity = capacity_of(i) - cpu_util(i);
if (spare_capacity > max_spare_capacity) {
        max_spare_capacity = spare_capacity;
        spare_group = group;
}
```

* * *

```
if (fit_group)
        return fit_group;

if (spare_group)
        return spare_group;
```

5.1.5 find_idlest_cpu

Android has:

```
if (task_fits_spare(p, i)) {
```

while Linux has:

```
if (idle_cpu(i)) {
```

* * *

Android has an extra nested *if* under task_fits_spare() *if*:

```
} else if (shallowest_idle_cpu == -1) {
        /*
         * If we haven't found an idle CPU yet
         * pick a non-idle one that can fit the task as
         * fallback.
         */
        shallowest_idle_cpu = i;
}
```

5.1.6 select_idle_sibling

New initialized variables in Android

```
struct sched_group *sg;
int i = task_cpu(p); //not initialized with a value in Linux
int best_idle = -1;
int best_idle_cstate = -1;
int best_idle_capacity = INT_MAX;
```

* * *

```
if (!sysctl_sched_cstate_aware) {
        if (idle_cpu(target))
                return target;
```

* * *

Android has:

```
if (i != target && cpus_share_cache(i, target) && idle_cpu(i))
        return i;
```

whereas Linux has:

```
if (prev != target && cpus_share_cache(prev, target) && idle_cpu(prev))
        return prev;
```

* * *

```
/*
 * Otherwise, iterate the domains and find an elegible idle cpu.
 */
sd = rcu_dereference(per_cpu(sd_llc, target));
for_each_lower_domain(sd) {
        sg = sd->groups;
        do {
                if (!cpumask_intersects(sched_group_cpus(sg),
                                tsk_cpus_allowed(p)))
                        goto next;


                if (sysctl_sched_cstate_aware) {
                        for_each_cpu_and(i, tsk_cpus_allowed(p), sched_group_cpus(sg\
                        )) {
                                struct rq *rq = cpu_rq(i);
                                int idle_idx = idle_get_state_idx(rq);
                                                                                      unsigned lon
                                unsigned long capacity_orig = capacity_orig_of(i);
                                if (new_usage > capacity_orig || !idle_cpu(i))
                                        goto next;

                                if (i == target && new_usage <= capacity_curr_of(tar\
                        get))
                                        return target;

                                if (best_idle < 0 || (idle_idx < best_idle_cstate &&\
                         capacity_orig <= best_idle_capacity)) {
                                                                                       best_idle =
                                        best_idle_cstate = idle_idx;
                                        best_idle_capacity = capacity_orig;
                                }
                        }
                } else {
                        for_each_cpu(i, sched_group_cpus(sg)) {
                                if (i == target || !idle_cpu(i))
                                        goto next;
                        }
                                                                target = cpumask_first_and(s
                                tsk_cpus_allowed(p));
                        goto done;
                }
next:
                sg = sg->next;
        } while (sg != sd->groups);
}
if (best_idle > 0)
        target = best_idle;

done:
        return target;
```

5.1.7 select_task_rq_fair

Android has:

```
want_affine = (!wake_wide(p) && task_fits_max(p, cpu) && cpumask_test_cpu(cpu, tsk_cpus_allo
```

whereas Linux has:

```
want_affine = !wake_wide(p) && !wake_cap(p, cpu, prev_cpu) && cpumask_test_cpu(cpu, tsk_cpus
```

* * *

New in *if (!sd)*:

```
if (energy_aware() && !cpu_rq(cpu)->rd->overutilized)
        new_cpu = energy_aware_wake_cpu(p, prev_cpu, sync);
```

5.1.8 pick_next_task_fair

This line occurs multiple times:

```
rq->misfit_task = !task_fits_max(p, rq->cpu);
```

5.1.9 struct lb_env

```
unsigned int            src_grp_nr_running;
```

5.1.10 detach_task

```
double_lock_balance(env->src_rq, env->dst_rq);
```

* * *

```
double_unlock_balance(env->src_rq, env->dst_rq);
```

5.1.11 attach_one_task

```
/*
 * We want to potentially raise target_cpu's OPP.
 */
update_capacity_of(cpu_of(rq));
```

5.1.12 attach_tasks

```
/*
 * We want to potentially raise env.dst_cpu's OPP.
 */
update_capacity_of(env->dst_cpu);
```

5.1.13 update_cpu_capacity

```
struct max_cpu_capacity *mcc;
unsigned long max_capacity;
int max_cap_cpu;
unsigned long flags;
```

* * *

```
        mcc = &cpu_rq(cpu)->rd->max_cpu_capacity;

        raw_spin_lock_irqsave(&mcc->lock, flags);
        max_capacity = mcc->val;
        max_cap_cpu = mcc->cpu;

        if ((max_capacity > capacity && max_cap_cpu == cpu) ||
            (max_capacity < capacity)) {
                mcc->val = capacity;
                mcc->cpu = cpu;
#ifdef CONFIG_SCHED_DEBUG
                raw_spin_unlock_irqrestore(&mcc->lock, flags);
                pr_info("CPU%d: update max cpu_capacity %lu\n", cpu, capacity);
                goto skip_unlock;
#endif
        }
        raw_spin_unlock_irqrestore(&mcc->lock, flags);

skip_unlock: __attribute__ ((unused));
```

* * *

```
sdg->sgc->max_capacity = capacity;
```

5.1.14 update_group_capacity

```
max_capacity = 0;
```

* * *

Android has:

```
if (unlikely(!rq->sd)) {
        capacity += capacity_of(cpu);
} else {
        sgc = rq->sd->groups->sgc;
        capacity += sgc->capacity;
}

max_capacity = max(capacity, max_capacity);
```

while Linux has:

```
if (unlikely(!rq->sd)) {
        capacity += capacity_of(cpu);
        continue;
}

sgc = rq->sd->groups->sgc;
capacity += sgc->capacity;
```

* * *

```
do {
        struct sched_group_capacity *sgc = group->sgc; //new in Android

        capacity += sgc->capacity;
        max_capacity = max(sgc->max_capacity, max_capacity); //new in Androi\
        d
        group = group->next;
} while (group != child->groups);
```

* * *

```
sdg->sgc->max_capacity = max_capacity;
```

5.1.15 update_sg_lb_stats

New argument:

```
@overutilized: Indicate overutilization for any CPU.
```

* * *

```
if (cpu_overutilized(i)) {
        *overutilized = true;
        if (!sgs->group_misfit_task && rq->misfit_task)
                sgs->group_misfit_task = capacity_of(i);
}
```

5.1.16 find_busiest_group

```
if (energy_aware() && !env->dst_rq->rd->overutilized)
                        goto out_balanced;
```

5.1.17 need_active_balance

```
if ((capacity_of(env->src_cpu) < capacity_of(env->dst_cpu)) &&
                        env->src_rq->cfs.h_nr_running == 1 &&
                        cpu_overutilized(env->src_cpu) &&
                        !cpu_overutilized(env->dst_cpu)) {
                return 1;
}
```

5.1.18 load_balance

```
/*
 * We want to potentially lower env.src_cpu's OPP.
 */
if (cur_ld_moved)
        update_capacity_of(env.src_cpu);
```

5.1.19 idle_balance

energyaware() is added to this *if* statement:

```
if (!energy_aware() && (this_rq->avg_idle < sysctl_sched_migration_cost || \
        !this_rq->rd->overload)) {
```

* * *

```
/*
 * If removed_util_avg is !0 we most probably migrated some task away
 * from this_cpu. In this case we might be willing to trigger an OPP
 * update, but we want to do so if we don't find anybody else to pull
 * here (we will trigger an OPP update with the pulled task's enqueue
 * anyway).
 *
 * Record removed_util before calling update_blocked_averages, and use
 * it below (before returning) to see if an OPP update is required.
 */
removed_util = atomic_long_read(&(this_rq->cfs).removed_util_avg);
```

* * *

```
else if (removed_util) {
        /*
         * No task pulled and someone has been migrated away.
         * Good case to trigger an OPP update.
         */
        update_capacity_of(this_cpu);
}
```

5.1.20 active_load_balance_cpu_stop

```
                                /*
                                 * We want to potentially lower env.src_cpu's OPP.
                                 */
                                update_capacity_of(env.src_cpu);
```

5.1.21 nohz_kick_needed

Second condition is new in Android:

```
if (rq->nr_running >= 2 && (!energy_aware() || cpu_overutilized(cpu)))
```

* * *

Second condition is new in Android:

```
if (sds && !energy_aware())
```

5.1.22 task_tick_fair

```
#ifdef CONFIG_SMP
        if (!rq->rd->overutilized && cpu_overutilized(task_cpu(curr))) {
                rq->rd->overutilized = true;
                trace_sched_overutilized(true);
        }

        rq->misfit_task = !task_fits_max(curr, rq->cpu);
#endif
```

5.2 core.c

5.2.1 move_queued_task

```
double_lock_balance(rq, cpu_rq(new_cpu));
```

* * *

```
double_unlock_balance(rq, cpu_rq(new_cpu));
```

5.2.2 set_task_cpu

```
walt_fixup_busy_time(p, new_cpu);
```

5.2.3 try_to_wake_up

```
#ifdef CONFIG_SMP
        struct rq *rq;
        u64 wallclock;
#endif
```

* * *

```
rq = cpu_rq(task_cpu(p));

        raw_spin_lock(&rq->lock);
        wallclock = walt_ktime_clock();
        walt_update_task_ravg(rq->curr, rq, TASK_UPDATE, wallclock, 0);
        walt_update_task_ravg(p, rq, TASK_WAKE, wallclock, 0);
        raw_spin_unlock(&rq->lock);
```

5.2.4 try_to_wake_up_local

```
u64 wallclock = walt_ktime_clock();

walt_update_task_ravg(rq->curr, rq, TASK_UPDATE, wallclock, 0);
walt_update_task_ravg(p, rq, TASK_WAKE, wallclock, 0);
```

5.2.5 __sched_fork

```
walt_init_new_task_load(p);
```

5.2.6 wake_up_new_task

```
walt_init_new_task_load(p);
```

* * *

```
walt_mark_task_starting(p);
```

* * *

In Android, the line

```
activate_task(rq, p, ENQUEUE_WAKEUP_NEW);
```

differs from the line in Linux:

```
activate_task(rq, p, 0);
```

5.2.7 scheduler_tick

```
walt_set_window_start(rq);
```

* * *

```
walt_update_task_ravg(rq->curr, rq, TASK_UPDATE, walt_ktime_clock(), 0);
```

* * *

```
sched_freq_tick(cpu);
```

5.2.8 __schedule

```
u64 wallclock;
```

* * *

```
wallclock = walt_ktime_clock();
walt_update_task_ravg(prev, rq, PUT_PREV_TASK, wallclock, 0);
walt_update_task_ravg(next, rq, PICK_NEXT_TASK, wallclock, 0);
```

5.2.9 sched_domain_debug_one

Android has

```
printk(KERN_CONT " (cpu_capacity = %lu)"
```

while Linux has

```
printk(KERN_CONT " (cpu_capacity = %d)",
```

5.2.10 sd_degenerate and sd_parent_degenerate

Android includes SD_SHARE_CAP_STATES check in an *if* statement in both functions.

5.2.11 init_rootdomain

```
init_max_cpu_capacity(&rd->max_cpu_capacity);
```

5.2.12 update_top_cache_domain

```
struct sched_domain *ea_sd = NULL;
```

* * *

```
for_each_domain(cpu, sd) {
        if (sd->groups->sge)
                ea_sd = sd;
        else
                break;
}
rcu_assign_pointer(per_cpu(sd_ea, cpu), ea_sd);

sd = highest_flag_domain(cpu, SD_SHARE_CAP_STATES);
rcu_assign_pointer(per_cpu(sd_scs, cpu), sd);
```

5.2.13 build_overlap_sched_groups

```
sg->sgc->max_capacity = SCHED_CAPACITY_SCALE;
```

5.2.14 build_sched_domains

```
struct sched_domain_topology_level *tl = sched_domain_topology;
```

* * *

In the code below, tl++ and the last lane are new in Android.

```
for (sd = *per_cpu_ptr(d.sd, i); sd; sd = sd->parent, tl++) {
                        init_sched_energy(i, sd, tl->energy);
```

* * *

Android doesn't use READ_ONCE() or WRITE_ONCE() (used in Linux to "avoid load/storing tearing"), and there are also a few debug lines only in Linux.

5.2.15 sched_cpu_dying

```
walt_migrate_sync_cpu(cpu);
```

5.2.16 sched_init_smp

```
walt_init_cpu_efficiency();
```

5.2.17 sched_init

```
#ifdef CONFIG_SCHED_WALT
                rq->cur_irqload = 0;
                rq->avg_irqload = 0;
                rq->irqload_ts = 0;
#endif
```

5.2.18 ___might_sleep

Android has

```
!is_idle_task(current)) || oops_in_progress)
            return;
    if (system_state != SYSTEM_RUNNING &&
        (!__might_sleep_init_called || system_state != SYSTEM_BOOTING))
```

while Linux has

```
 !is_idle_task(current)) ||
system_state != SYSTEM_RUNNING || oops_in_progress)
    return;
```

5.3 rt.c

5.3.1 sched_rt_runtime_exceeded

```
static bool once = false;
```

* * *

```
if (!once) {
      once = true;
      dump_throttled_rt_tasks(rt_rq);
}
```

5.4 sched.h

5.4.1 max_cpu_capacity (new)

```
struct max_cpu_capacity {
    raw_spinlock_t lock;
    unsigned long val;
    int cpu;
};
```

5.4.2 cfs_rq

```
#ifdef CONFIG_SCHED_WALT
    u64 cumulative_runnable_avg;
#endif
```

5.4.3 root_domain

```
/* Indicate one or more cpus over-utilized (tipping point) */
bool overutilized;
```

* * *

This replaces *unsigned long max_cpu_capacity*:

```
struct max_cpu_capacity max_cpu_capacity;
```

5.4.4 rq

```
unsigned int misfit_task;
```

* * *

```
  #ifdef CONFIG_CPU_QUIET
    /* time-based average load */
    u64 nr_last_stamp;
    u64 nr_running_integral;
    seqcount_t ave_seqcnt;
  #endif
```

* * *

```
  #ifdef CONFIG_SCHED_WALT
    /*
     * max_freq = user or thermal defined maximum
     * max_possible_freq = maximum supported by hardware
     */
    unsigned int cur_freq, max_freq, min_freq, max_possible_freq;
    struct cpumask freq_domain_cpumask;

    u64 cumulative_runnable_avg;
    int efficiency; /* Differentiate cpus with different IPC capability */
    int load_scale_factor;
    int capacity;
    int max_possible_capacity;
    u64 window_start;
    u64 curr_runnable_sum;
    u64 prev_runnable_sum;
    u64 nt_curr_runnable_sum;
    u64 nt_prev_runnable_sum;
    u64 cur_irqload;
    u64 avg_irqload;
    u64 irqload_ts;
  #endif /* CONFIG_SCHED_WALT */
```

* * *

In *#ifdef CONFIG_CPU_IDLE*:

```
int idle_state_idx;
```

5.4.5 DECLARE_PER_CPU

```
DECLARE_PER_CPU(struct sched_domain *, sd_ea);
```

* * *

```
DECLARE_PER_CPU(struct sched_domain *, sd_scs);
```

5.4.6 sched_group_capacity

```
unsigned long max_capacity; /* Max per-cpu capacity in group */
```

5.4.7 sched_group

```
const struct sched_group_energy *sge;
```

5.5 cputime.c

5.5.1 irqtime_account_irq

```
#ifdef CONFIG_SCHED_WALT
        u64 wallclock;
        bool account = true;
#endif
```

* * *

```
#ifdef CONFIG_SCHED_WALT
        wallclock = sched_clock_cpu(cpu);
#endif
```

* * *

```
#ifdef CONFIG_SCHED_WALT
        else
                account = false;
#endif
```

* * *

```
#ifdef CONFIG_SCHED_WALT
        if (account)
                walt_account_irqtime(cpu, curr, \
delta, wallclock);
#endif
```

5.6 idle.c

5.6.1 sched_idle_set_state

There is a new argument: *int index*

* * *

```
idle_set_state_idx(this_rq(), index);
```

5.7 stop_task.c

5.7.1 enqueue_task_stop

```
walt_inc_cumulative_runnable_avg(rq, p);
```

5.7.2 dequeue_task_stop

```
walt_dec_cumulative_runnable_avg(rq, p);
```

5.8 features.h

```
/*
 * Energy aware scheduling. Use platform energy \
model to guide scheduling
 * decisions optimizing for energy efficiency.
 */
#ifdef CONFIG_DEFAULT_USE_ENERGY_AWARE
SCHED_FEAT(ENERGY_AWARE, true)
#else
SCHED_FEAT(ENERGY_AWARE, false)
#endif
```

6 Function Definitions

6.1 fair.c

6.1.1 energy_diff

```
static inline int
energy_diff(struct energy_env *eenv)
{
    unsigned int boost;
    int nrg_delta;

    /* Conpute "absolute" energy diff */
    __energy_diff(eenv);
    /* Return energy diff when boost margin is 0 */
#ifdef CONFIG_CGROUP_SCHEDTUNE
    boost = schedtune_task_boost(eenv->task);
#else
    boost = get_sysctl_sched_cfs_boost();
#endif
    if (boost == 0)
            return eenv->nrg.diff;

    /* Compute normalized energy diff */
    nrg_delta = normalize_energy(eenv->nrg.diff);
    eenv->nrg.delta = nrg_delta;

    eenv->payoff = schedtune_accept_deltas(
                    eenv->nrg.delta,
                    eenv->cap.delta,
                    eenv->task);

    /*
     * When SchedTune is enabled, the energy_diff() function will return
     * the computed energy payoff value. Since the energy_diff() return
     * value is expected to be negative by its callers, this evaluation
     * function return a negative value each time the evaluation return a
     * positive payoff, which is the condition for the acceptance of
     * a scheduling decision
     */
    return -eenv->payoff;
}
```

6.1.2 __energy_diff

```
/*
* energy_diff(): Estimate the energy impact of changing the utilization
* distribution. eenv specifies the change: utilisation amount, source, and
* destination cpu. Source or destination cpu may be -1 in which case the
* utilization is removed from or added to the system (e.g. task wake-up). If
* both are specified, the utilization is migrated.
*/
static inline int __energy_diff(struct energy_env *eenv)
{
    struct sched_domain *sd;
    struct sched_group *sg;
    int sd_cpu = -1, energy_before = 0, energy_after = 0;

    struct energy_env eenv_before = {
            .util_delta     = 0,
            .src_cpu        = eenv->src_cpu,
            .dst_cpu        = eenv->dst_cpu,
            .nrg            = { 0, 0, 0, 0},
            .cap            = { 0, 0, 0 },
    };

    if (eenv->src_cpu == eenv->dst_cpu)
            return 0;

    sd_cpu = (eenv->src_cpu != -1) ? eenv->src_cpu : eenv->dst_cpu;
    sd = rcu_dereference(per_cpu(sd_ea, sd_cpu));

    if (!sd)
            return 0; /* Error */

    sg = sd->groups;

    do {
            if (cpu_in_sg(sg, eenv->src_cpu) || cpu_in_sg(sg, eenv->dst_cpu)) {
                    eenv_before.sg_top = eenv->sg_top = sg;

                    if (sched_group_energy(&eenv_before))
                            return 0; /* Invalid result abort */
                    energy_before += eenv_before.energy;

                    /* Keep track of SRC cpu (before) capacity */
                    eenv->cap.before = eenv_before.cap.before;
                    eenv->cap.delta = eenv_before.cap.delta;

                    if (sched_group_energy(eenv))
                            return 0; /* Invalid result abort */
                    energy_after += eenv->energy;
            }
    } while (sg = sg->next, sg != sd->groups);

    eenv->nrg.before = energy_before;
    eenv->nrg.after = energy_after;
    eenv->nrg.diff = eenv->nrg.after - eenv->nrg.before;
    eenv->payoff = 0;

    trace_sched_energy_diff(eenv->task,
                    eenv->src_cpu, eenv->dst_cpu, eenv->util_delta,
                    eenv->nrg.before, eenv->nrg.after, eenv->nrg.diff,
                    eenv->cap.before, eenv->cap.after, eenv->cap.delta,
                    eenv->nrg.delta, eenv->payoff);
```

```
    return eenv->nrg.diff;
}
```

6.1.3 update_capacity_of

```
static void update_capacity_of(int cpu)
{
    unsigned long req_cap;

    if (!sched_freq())
            return;

    /* Convert scale-invariant capacity to cpu. */
    req_cap = boosted_cpu_util(cpu);
    req_cap = req_cap * SCHED_CAPACITY_SCALE / capacity_orig_of(cpu);
    set_cfs_cpu_capacity(cpu, true, req_cap);
}
```

6.1.4 capacity_curr_of

```
unsigned long capacity_curr_of(int cpu)
{
    return cpu_rq(cpu)->cpu_capacity_orig *
           arch_scale_freq_capacity(NULL, cpu)
           >> SCHED_CAPACITY_SHIFT;
}
```

6.1.5 energy_aware

```
static inline bool energy_aware(void)
{
    return sched_feat(ENERGY_AWARE);
}
```

6.1.6 __cpu_norm_util

```
/*
* __cpu_norm_util() returns the cpu util relative to a specific capacity,
* i.e. it's busy ratio, in the range [0..SCHED_LOAD_SCALE] which is useful for
* energy calculations. Using the scale-invariant util returned by
* cpu_util() and approximating scale-invariant util by:
*
*   util ~ (curr_freq/max_freq)*1024 * capacity_orig/1024 * running_time/time
*
* the normalized util can be found using the specific capacity.
*
*   capacity = capacity_orig * curr_freq/max_freq
```

```
*
*   norm_util = running_time/time ~ util/capacity
*/
static unsigned long __cpu_norm_util(int cpu, unsigned long capacity, int delta)
{
    int util = __cpu_util(cpu, delta);

    if (util >= capacity)
            return SCHED_CAPACITY_SCALE;

    return (util << SCHED_CAPACITY_SHIFT)/capacity;
}
```

6.1.7 calc_util_delta

```
static int calc_util_delta(struct energy_env *eenv, int cpu)
{
    if (cpu == eenv->src_cpu)
            return -eenv->util_delta;
    if (cpu == eenv->dst_cpu)
            return eenv->util_delta;
    return 0;
}
```

6.1.8 group_max_util

```
/*
* group_norm_util() returns the approximated group util relative to it's
* current capacity (busy ratio) in the range [0..SCHED_LOAD_SCALE] for use in
* energy calculations. Since task executions may or may not overlap in time in
* the group the true normalized util is between max(cpu_norm_util(i)) and
* sum(cpu_norm_util(i)) when iterating over all cpus in the group, i. The
* latter is used as the estimate as it leads to a more pessimistic energy
* estimate (more busy).
*/
static unsigned
long group_norm_util(struct energy_env *eenv, struct sched_group *sg)
{
    int i, delta;
    unsigned long util_sum = 0;
    unsigned long capacity = sg->sge->cap_states[eenv->cap_idx].cap;

    for_each_cpu(i, sched_group_cpus(sg)) {
            delta = calc_util_delta(eenv, i);
            util_sum += __cpu_norm_util(i, capacity, delta);
    }
    if (util_sum > SCHED_CAPACITY_SCALE)
            return SCHED_CAPACITY_SCALE;
    return util_sum;
}
```

6.1.9 find_new_capacity

```
static int find_new_capacity(struct energy_env *eenv,
    const struct sched_group_energy * const sge)
{
    int idx;
    unsigned long util = group_max_util(eenv);

    for (idx = 0; idx < sge->nr_cap_states; idx++) {
            if (sge->cap_states[idx].cap >= util)
                    break;
    }

    eenv->cap_idx = idx;

    return idx;
}
```

6.1.10 group_idle_state

```
static int group_idle_state(struct sched_group *sg)
{
    int i, state = INT_MAX;

    /* Find the shallowest idle state in the sched group. */
    for_each_cpu(i, sched_group_cpus(sg))
            state = min(state, idle_get_state_idx(cpu_rq(i)));

    /* Take non-cpuidle idling into account (active idle/arch_cpu_idle()) */
    state++;

    return state;
}
```

6.1.11 sched_group_energy

```
/*
* sched_group_energy(): Computes the absolute energy consumption of cpus
* belonging to the sched_group including shared resources shared only by
* members of the group. Iterates over all cpus in the hierarchy below the
* sched_group starting from the bottom working it's way up before going to
* the next cpu until all cpus are covered at all levels. The current
* implementation is likely to gather the same util statistics multiple times.
* This can probably be done in a faster but more complex way.
* Note: sched_group_energy() may fail when racing with sched_domain updates.
*/
static int sched_group_energy(struct energy_env *eenv)
{
    struct sched_domain *sd;
    int cpu, total_energy = 0;
    struct cpumask visit_cpus;
    struct sched_group *sg;

    WARN_ON(!eenv->sg_top->sge);

    cpumask_copy(&visit_cpus, sched_group_cpus(eenv->sg_top));

    while (!cpumask_empty(&visit_cpus)) {
            struct sched_group *sg_shared_cap = NULL;

            cpu = cpumask_first(&visit_cpus);

            /*
             * Is the group utilization affected by cpus outside this
             * sched_group?
             */
            sd = rcu_dereference(per_cpu(sd_scs, cpu));

            if (!sd)
                    /*
                     * We most probably raced with hotplug; returning a
                     * wrong energy estimation is better than entering an
                     * infinite loop.
                     */
                    return -EINVAL;

            if (sd->parent)
                    sg_shared_cap = sd->parent->groups;

            for_each_domain(cpu, sd) {
                    sg = sd->groups;

                    /* Has this sched_domain already been visited? */
                    if (sd->child && group_first_cpu(sg) != cpu)
                            break;
                    do {
                            unsigned long group_util;
                            int sg_busy_energy, sg_idle_energy;
                            int cap_idx, idle_idx;

                            if (sg_shared_cap && sg_shared_cap->group_weight >= sg->grou$
                                    eenv->sg_cap = sg_shared_cap;
                            else
                                    eenv->sg_cap = sg;

                            cap_idx = find_new_capacity(eenv, sg->sge);

                            if (sg->group_weight == 1) {
                                    /* Remove capacity of src CPU (before task move) */
                                    if (eenv->util_delta == 0 &&
                                        cpumask_test_cpu(eenv->src_cpu, sched_group_cpus$
                                            eenv->cap.before = sg->sge->cap_states[cap_i$
                                            eenv->cap.delta -= eenv->cap.before;
                                    }
                            }

                            idle_idx = group_idle_state(sg);
                            group_util = group_norm_util(eenv, sg);
                            sg_busy_energy = (group_util * sg->sge->cap_states[cap_idx].$
                                                            >> SCHED_CAPACITY_SHIFT;
                            sg_idle_energy = ((SCHED_CAPACITY_SCALE-group_util)
                                                            * sg->sge->idle_states[idle_$
                                                            >> SCHED_CAPACITY_SHIFT;

                            total_energy += sg_busy_energy + sg_idle_energy;

                            if (!sd->child)
                                    cpumask_xor(&visit_cpus, &visit_cpus, sched_group_cp$

                            if (cpumask_equal(sched_group_cpus(sg), sched_group_cpus(een$
                                    goto next_cpu;

                    } while (sg = sg->next, sg != sd->groups);
            }
next_cpu:
            cpumask_clear_cpu(cpu, &visit_cpus);
            continue;
    }

    eenv->energy = total_energy;
    return 0;
}
```

6.1.12 cpu_in_sg

```
static inline bool cpu_in_sg(struct sched_group *sg, int cpu)
{
    return cpu != -1 && cpumask_test_cpu(cpu, sched_group_cpus(sg));
}
```

6.1.13 normalize_energy

```
/*
* System energy normalization
* Returns the normalized value, in the range [0..SCHED_LOAD_SCALE],
* corresponding to the specified energy variation.
*/
static inline int
normalize_energy(int energy_diff)
{
    u32 normalized_nrg;
#ifdef CONFIG_SCHED_DEBUG
    int max_delta;

    /* Check for boundaries */
    max_delta  = schedtune_target_nrg.max_power;
    max_delta -= schedtune_target_nrg.min_power;
    WARN_ON(abs(energy_diff) >= max_delta);
#endif

    /* Do scaling using positive numbers to increase the range */
    normalized_nrg = (energy_diff < 0) ? -energy_diff : energy_diff;

    /* Scale by energy magnitude */
    normalized_nrg <<= SCHED_CAPACITY_SHIFT;

    /* Normalize on max energy for target platform */
    normalized_nrg = reciprocal_divide(
                normalized_nrg, schedtune_target_nrg.rdiv);

    return (energy_diff < 0) ? -normalized_nrg : normalized_nrg;
}
```

6.1.14 boosted_task_util

```
static inline unsigned long
boosted_task_util(struct task_struct *task)
```

```c
{
    unsigned long util = task_util(task);
    long margin = schedtune_task_margin(task);

    trace_sched_boost_task(task, util, margin);

    return util + margin;
}
```

6.1.15 __task_fits

```c
static inline bool __task_fits(struct task_struct *p, int cpu, int util)
{
    unsigned long capacity = capacity_of(cpu);

    util += boosted_task_util(p);

    return (capacity * 1024) > (util * capacity_margin);
}
```

6.1.16 task_fits_max

```c
static inline bool task_fits_max(struct task_struct *p, int cpu)
{
    unsigned long capacity = capacity_of(cpu);
    unsigned long max_capacity = cpu_rq(cpu)->rd->max_cpu_capacity.val;

    if (capacity == max_capacity)
            return true;

    if (capacity * capacity_margin > max_capacity * 1024)
            return true;

    return __task_fits(p, cpu, 0);
}
```

6.1.17 task_fits_spare

```c
static inline bool task_fits_spare(struct task_struct *p, int cpu)
{
    return __task_fits(p, cpu, cpu_util(cpu));
}
```

6.1.18 schedtune_cpu_margin

```c
static inline int
schedtune_cpu_margin(unsigned long util, int cpu)
```

```
{
        int boost;

#ifdef CONFIG_CGROUP_SCHEDTUNE
        boost = schedtune_cpu_boost(cpu);
#else
        boost = get_sysctl_sched_cfs_boost();
#endif
        if (boost == 0)
                return 0;

        return schedtune_margin(util, boost);
}
```

6.1.19 cpu_overutilized

```
static bool cpu_overutilized(int cpu)
{
    return (capacity_of(cpu) * 1024) < (cpu_util(cpu) * capacity_margin);
}
```

6.1.20 schedtune_margin

```
static long
schedtune_margin(unsigned long signal, long boost)
{
    long long margin = 0;

    /*
     * Signal proportional compensation (SPC)
     *
     * The Boost (B) value is used to compute a Margin (M) which is
     * proportional to the complement of the original Signal (S):
     *   M = B * (SCHED_LOAD_SCALE - S), if B is positive
     *   M = B * S, if B is negative
     * The obtained M could be used by the caller to "boost" S.
     */

    if (boost >= 0) {
            margin  = SCHED_CAPACITY_SCALE - signal;
            margin *= boost;
    } else
            margin = -signal * boost;
    /*
     * Fast integer division by constant:
     *  Constant   :                 (C) = 100
```

```
	 *  Precision  : 0.1%             (P) = 0.1
	 *  Reference  : C * 100 / P      (R) = 100000
	 *
	 * Thus:
	 *  Shift bits : ceil(log(R,2))   (S) = 17
	 *  Mult const : round(2^S/C)     (M) = 1311
	 *
	 *
	 */
	margin  *= 1311;
	margin >>= 17;

	if (boost < 0)
		margin *= -1;
	return margin;
}
```

6.1.21 schedtune_cpu_margin

```
static inline int
schedtune_cpu_margin(unsigned long util, int cpu)
{
	int boost;

#ifdef CONFIG_CGROUP_SCHEDTUNE
	boost = schedtune_cpu_boost(cpu);
#else
	boost = get_sysctl_sched_cfs_boost();
#endif
	if (boost == 0)
		return 0;

	return schedtune_margin(util, boost);
}
```

6.1.22 schedtune_task_margin

```
static inline long
schedtune_task_margin(struct task_struct *task)
{
	int boost;
	unsigned long util;
	long margin;

#ifdef CONFIG_CGROUP_SCHEDTUNE
	boost = schedtune_task_boost(task);
#else
    boost = get_sysctl_sched_cfs_boost();
#endif
    if (boost == 0)
            return 0;

    util = task_util(task);
    margin = schedtune_margin(util, boost);

    return margin;
}
```

6.1.23 boosted_cpu_util

```
static inline unsigned long
boosted_cpu_util(int cpu)
{
    unsigned long util = cpu_util(cpu);
    long margin = schedtune_cpu_margin(util, cpu);

    trace_sched_boost_cpu(cpu, util, margin);

    return util + margin;
}
```

6.1.24 boosted_task_util

```
static inline unsigned long
boosted_task_util(struct task_struct *task)
{
    unsigned long util = task_util(task);
    long margin = schedtune_task_margin(task);

    trace_sched_boost_task(task, util, margin);

    return util + margin;
}
```

6.1.25 energy_aware_wake_cpu

```
static int energy_aware_wake_cpu(struct task_struct *p, int target, int sync)
{
    struct sched_domain *sd;
    struct sched_group *sg, *sg_target;
    int target_max_cap = INT_MAX;
    int target_cpu = task_cpu(p);
```

```
unsigned long task_util_boosted, new_util;
int i;

if (sysctl_sched_sync_hint_enable && sync) {
        int cpu = smp_processor_id();
        cpumask_t search_cpus;
        cpumask_and(&search_cpus, tsk_cpus_allowed(p), cpu_online_mask);
        if (cpumask_test_cpu(cpu, &search_cpus))
                return cpu;
}

sd = rcu_dereference(per_cpu(sd_ea, task_cpu(p)));

if (!sd)
        return target;

sg = sd->groups;
sg_target = sg;

if (sysctl_sched_is_big_little) {

        /*
         * Find group with sufficient capacity. We only get here if no cpu is
         * overutilized. We may end up overutilizing a cpu by adding the task,
         * but that should not be any worse than select_idle_sibling().
         * load_balance() should sort it out later as we get above the tipping
         * point.
         */
        do {
                /* Assuming all cpus are the same in group */
                int max_cap_cpu = group_first_cpu(sg);

                /*
                 * Assume smaller max capacity means more energy-efficient.
                 * Ideally we should query the energy model for the right
                 * answer but it easily ends up in an exhaustive search.
                 */
                if (capacity_of(max_cap_cpu) < target_max_cap &&
                    task_fits_max(p, max_cap_cpu)) {
                        sg_target = sg;
                        target_max_cap = capacity_of(max_cap_cpu);
                }
        } while (sg = sg->next, sg != sd->groups);

        task_util_boosted = boosted_task_util(p);
        /* Find cpu with sufficient capacity */
```

```
            for_each_cpu_and(i, tsk_cpus_allowed(p), sched_group_cpus(sg_target)) {
                    /*
                     * p's blocked utilization is still accounted for on prev_cpu
                     * so prev_cpu will receive a negative bias due to the double
                     * accounting. However, the blocked utilization may be zero.
                     */
                    new_util = cpu_util(i) + task_util_boosted;

                    /*
                     * Ensure minimum capacity to grant the required boost.
                     * The target CPU can be already at a capacity level higher
                     * than the one required to boost the task.
                     */
                    if (new_util > capacity_orig_of(i))
                            continue;

                    if (new_util < capacity_curr_of(i)) {
                            target_cpu = i;
            }
    } else {
            /*
             * Find a cpu with sufficient capacity
             */
#ifdef CONFIG_CGROUP_SCHEDTUNE
            bool boosted = schedtune_task_boost(p) > 0;
            bool prefer_idle = schedtune_prefer_idle(p) > 0;
#else
            bool boosted = 0;
            bool prefer_idle = 0;
#endif
            int tmp_target = find_best_target(p, boosted, prefer_idle);
            if (tmp_target >= 0) {
                    target_cpu = tmp_target;
                    if ((boosted || prefer_idle) && idle_cpu(target_cpu))
                            return target_cpu;
            }
    }

    if (target_cpu != task_cpu(p)) {
            struct energy_env eenv = {
                    .util_delta     = task_util(p),
                    .src_cpu        = task_cpu(p),
                    .dst_cpu        = target_cpu,
                    .task           = p,
            };
```

```
                /* Not enough spare capacity on previous cpu */
                if (cpu_overutilized(task_cpu(p)))
                        return target_cpu;

                if (energy_diff(&eenv) >= 0)
                        return task_cpu(p);
        }

        return target_cpu;
}
```

6.1.26 init_max_cpu_capacity

```
void init_max_cpu_capacity(struct max_cpu_capacity *mcc)
{
    raw_spin_lock_init(&mcc->lock);
    mcc->val = 0;
    mcc->cpu = -1;
}
```

6.2 walt.c

```
void
walt_inc_cumulative_runnable_avg(struct rq *rq,
                                 struct task_struct *p)
{
        rq->cumulative_runnable_avg += p->ravg.demand;
}
```

* * *

```
void
walt_dec_cumulative_runnable_avg(struct rq *rq,
                                 struct task_struct *p)
{
        rq->cumulative_runnable_avg -= p->ravg.demand;
        BUG_ON((s64)rq->cumulative_runnable_avg < 0);
}
```

* * *

```
static void
fixup_cumulative_runnable_avg(struct rq *rq,
                              struct task_struct *p, s64 task_load_delta)
{
        rq->cumulative_runnable_avg += task_load_delta;
        if ((s64)rq->cumulative_runnable_avg < 0)
```

```
                panic("cra less than zero: tld: %lld, task_load(p) = %u\n",
                        task_load_delta, task_load(p));
}
```

* * *

```
u64 walt_ktime_clock(void)
{
        if (unlikely(walt_ktime_suspended))
                return ktime_to_ns(ktime_last);
        return ktime_get_ns();
}
```

* * *

```
static void walt_resume(void)
{
        walt_ktime_suspended = false;
}
```

* * *

```
static int walt_suspend(void)
{
        ktime_last = ktime_get();
        walt_ktime_suspended = true;
        return 0;
}
```

* * *

```
static int __init walt_init_ops(void)
{
        register_syscore_ops(&walt_syscore_ops);
        return 0;
}
```

* * *

```
void walt_inc_cfs_cumulative_runnable_avg(struct cfs_rq *cfs_rq,
                struct task_struct *p)
{
        cfs_rq->cumulative_runnable_avg += p->ravg.demand;
}
```

* * *

```
void walt_dec_cfs_cumulative_runnable_avg(struct cfs_rq *cfs_rq,
                struct task_struct *p)
{
        cfs_rq->cumulative_runnable_avg -= p->ravg.demand;
}
```

* * *

```
static int exiting_task(struct task_struct *p)
{
        if (p->flags & PF_EXITING) {
                if (p->ravg.sum_history[0] != EXITING_TASK_MARKER) {
                        p->ravg.sum_history[0] = EXITING_TASK_MARKER;
                }
                return 1;
        }
        return 0;
}
```

* * *

```
static int __init set_walt_ravg_window(char *str)
{
        get_option(&str, &walt_ravg_window);

        walt_disabled = (walt_ravg_window < MIN_SCHED_RAVG_WINDOW ||
                                walt_ravg_window > MAX_SCHED_RAVG_WINDOW);
        return 0;
}
```

* * *

```
static void
update_window_start(struct rq *rq, u64 wallclock)
{
        s64 delta;
        int nr_windows;

        delta = wallclock - rq->window_start;
        /* If the MPM global timer is cleared, set delta as 0 to avoid kernel BUG happening
        if (delta < 0) {
                delta = 0;
                WARN_ONCE(1, "WALT wallclock appears to have gone backwards or reset\n");
        }

        if (delta < walt_ravg_window)
                return;
        nr_windows = div64_u64(delta, walt_ravg_window);
        rq->window_start += (u64)nr_windows * (u64)walt_ravg_window;
}
```

* * *

```
static u64 scale_exec_time(u64 delta, struct rq *rq)
{
        unsigned int cur_freq = rq->cur_freq;
        int sf;

        if (unlikely(cur_freq > max_possible_freq))
                cur_freq = rq->max_possible_freq;

        /* round up div64 */
        delta = div64_u64(delta * cur_freq + max_possible_freq - 1,
                          max_possible_freq);

        sf = DIV_ROUND_UP(rq->efficiency * 1024, max_possible_efficiency);

        delta *= sf;
        delta >>= 10;

        return delta;
}
```

* * *

```
static int cpu_is_waiting_on_io(struct rq *rq)
{
        if (!walt_io_is_busy)
                return 0;

        return atomic_read(&rq->nr_iowait);
}
```

* * *

```
void walt_account_irqtime(int cpu, struct task_struct *curr,
                                 u64 delta, u64 wallclock)
{
        struct rq *rq = cpu_rq(cpu);
        unsigned long flags, nr_windows;
        u64 cur_jiffies_ts;

        raw_spin_lock_irqsave(&rq->lock, flags);
        /*
         * cputime (wallclock) uses sched_clock so use the same here for
         * consistency.
         */
        delta += sched_clock() - wallclock;
        cur_jiffies_ts = get_jiffies_64();

        if (is_idle_task(curr))
                walt_update_task_ravg(curr, rq, IRQ_UPDATE, walt_ktime_clock(),
                                 delta);

        nr_windows = cur_jiffies_ts - rq->irqload_ts;

        if (nr_windows) {
                if (nr_windows < 10) {
                        /* Decay CPU's irqload by 3/4 for each window. */
                        rq->avg_irqload *= (3 * nr_windows);
                        rq->avg_irqload = div64_u64(rq->avg_irqload,
                                                    4 * nr_windows);
                } else {
                        rq->avg_irqload = 0;
                }
                rq->avg_irqload += rq->cur_irqload;
                rq->cur_irqload = 0;
                        }

        rq->cur_irqload += delta;
        rq->irqload_ts = cur_jiffies_ts;
        raw_spin_unlock_irqrestore(&rq->lock, flags);
}
```

* * *

```
u64 walt_irqload(int cpu) {
        struct rq *rq = cpu_rq(cpu);
        s64 delta;
        delta = get_jiffies_64() - rq->irqload_ts;

        /*
         * Current context can be preempted by irq and rq->irqload_ts can be
         * updated by irq context so that delta can be negative.
         * But this is okay and we can safely return as this means there
         * was recent irq occurrence.
         */
        if (delta < WALT_HIGH_IRQ_TIMEOUT)
                return rq->avg_irqload;
                        else
                return 0;
}
```

* * *

```
int walt_cpu_high_irqload(int cpu) {
        return walt_irqload(cpu) >= sysctl_sched_walt_cpu_high_irqload;
}
```

* * *

```
static int account_busy_for_cpu_time(struct rq *rq, struct task_struct *p,
                                     u64 irqtime, int event)
{
        if (is_idle_task(p)) {
                /* TASK_WAKE && TASK_MIGRATE is not possible on idle task! */
                if (event == PICK_NEXT_TASK)
                        return 0;

                /* PUT_PREV_TASK, TASK_UPDATE && IRQ_UPDATE are left */
                return irqtime || cpu_is_waiting_on_io(rq);
        }

        if (event == TASK_WAKE)
                return 0;

        if (event == PUT_PREV_TASK || event == IRQ_UPDATE ||
                                         event == TASK_UPDATE)
                return 1;

        /* Only TASK_MIGRATE && PICK_NEXT_TASK left */
        return walt_freq_account_wait_time;
}
```

* * *

```
/*
 * Account cpu activity in its busy time counters (rq->curr/prev_runnable_sum)
 */
static void update_cpu_busy_time(struct task_struct *p, struct rq *rq,
             int event, u64 wallclock, u64 irqtime)
{
        int new_window, nr_full_windows = 0;
        int p_is_curr_task = (p == rq->curr);
```

```
u64 mark_start = p->ravg.mark_start;
u64 window_start = rq->window_start;
u32 window_size = walt_ravg_window;
u64 delta;

new_window = mark_start < window_start;
if (new_window) {
        nr_full_windows = div64_u64((window_start - mark_start),
                                        window_size);
        if (p->ravg.active_windows < USHRT_MAX)
                p->ravg.active_windows++;
                        }

/* Handle per-task window rollover. We don't care about the idle
 * task or exiting tasks. */
if (new_window && !is_idle_task(p) && !exiting_task(p)) {
        u32 curr_window = 0;

        if (!nr_full_windows)
                curr_window = p->ravg.curr_window;

        p->ravg.prev_window = curr_window;
        p->ravg.curr_window = 0;
                }

if (!account_busy_for_cpu_time(rq, p, irqtime, event)) {
        /* account_busy_for_cpu_time() = 0, so no update to the
         * task's current window needs to be made. This could be
         * for example
         *
         *   - a wakeup event on a task within the current
         *     window (!new_window below, no action required),
         *   - switching to a new task from idle (PICK_NEXT_TASK)
         *     in a new window where irqtime is 0 and we aren't
         *     waiting on IO */

        if (!new_window)
                return;

        /* A new window has started. The RQ demand must be rolled
         * over if p is the current task. */
        if (p_is_curr_task) {
                u64 prev_sum = 0;

                /* p is either idle task or an exiting task */
                if (!nr_full_windows) {
```

```
                                prev_sum = rq->curr_runnable_sum;
                                                        }

                        rq->prev_runnable_sum = prev_sum;
                        rq->curr_runnable_sum = 0;
                }

                return;
        }

        if (!new_window) {
                /* account_busy_for_cpu_time() = 1 so busy time needs
                 * to be accounted to the current window. No rollover
                 * since we didn't start a new window. An example of this is
                 * when a task starts execution and then sleeps within the
                 * same window. */

                if (!irqtime || !is_idle_task(p) || cpu_is_waiting_on_io(rq))
                        delta = wallclock - mark_start;
                else
                        delta = irqtime;
                delta = scale_exec_time(delta, rq);
                rq->curr_runnable_sum += delta;
                if (!is_idle_task(p) && !exiting_task(p))
                        p->ravg.curr_window += delta;

                return;
        }

        if (!p_is_curr_task) {
                /* account_busy_for_cpu_time() = 1 so busy time needs
                 * to be accounted to the current window. A new window
                 * has also started, but p is not the current task, so the
                 * window is not rolled over - just split up and account
                 * as necessary into curr and prev. The window is only
                 * rolled over when a new window is processed for the current
                 * task.
                 *
                 * Irqtime can't be accounted by a task that isn't the
                 * currently running task. */

                if (!nr_full_windows) {
                        /* A full window hasn't elapsed, account partial
                         * contribution to previous completed window. */
                        delta = scale_exec_time(window_start - mark_start, rq);
                        if (!exiting_task(p))
```

```
                                p->ravg.prev_window += delta;
                } else {
                        /* Since at least one full window has elapsed,
                         * the contribution to the previous window is the
                         * full window (window_size). */
                        delta = scale_exec_time(window_size, rq);
                        if (!exiting_task(p))
                                p->ravg.prev_window = delta;
                }
                rq->prev_runnable_sum += delta;

                /* Account piece of busy time in the current window. */
                delta = scale_exec_time(wallclock - window_start, rq);
                rq->curr_runnable_sum += delta;
                if (!exiting_task(p))
                        p->ravg.curr_window = delta;

                return;
        }

        if (!irqtime || !is_idle_task(p) || cpu_is_waiting_on_io(rq)) {
                /* account_busy_for_cpu_time() = 1 so busy time needs
                 * to be accounted to the current window. A new window
                 * has started and p is the current task so rollover is
                 * needed. If any of these three above conditions are true
                 * then this busy time can't be accounted as irqtime.
                 *
                  * Busy time for the idle task or exiting tasks need not
                 * be accounted.
                 *
                 * An example of this would be a task that starts execution
                 * and then sleeps once a new window has begun. */

                if (!nr_full_windows) {
                        /* A full window hasn't elapsed, account partial
                         * contribution to previous completed window. */
                        delta = scale_exec_time(window_start - mark_start, rq);
                        if (!is_idle_task(p) && !exiting_task(p))
                                p->ravg.prev_window += delta;

                        delta += rq->curr_runnable_sum;
                } else {
                        /* Since at least one full window has elapsed,
                         * the contribution to the previous window is the
                         * full window (window_size). */
                        delta = scale_exec_time(window_size, rq);
                        if (!is_idle_task(p) && !exiting_task(p))
                                p->ravg.prev_window = delta;

                }
                /*
                 * Rollover for normal runnable sum is done here by overwriting
                 * the values in prev_runnable_sum and curr_runnable_sum.
                 * Rollover for new task runnable sum has completed by previous
                 * if-else statement.
                 */
                rq->prev_runnable_sum = delta;

                /* Account piece of busy time in the current window. */
                delta = scale_exec_time(wallclock - window_start, rq);
                rq->curr_runnable_sum = delta;
                if (!is_idle_task(p) && !exiting_task(p))
                        p->ravg.curr_window = delta;

                return;
        }

        if (irqtime) {
                /* account_busy_for_cpu_time() = 1 so busy time needs
                 * to be accounted to the current window. A new window
                 * has started and p is the current task so rollover is
                 * needed. The current task must be the idle task because
                 * irqtime is not accounted for any other task.
                 *
                 * Irqtime will be accounted each time we process IRQ activity
                 * after a period of idleness, so we know the IRQ busy time
                 * started at wallclock - irqtime. */

                BUG_ON(!is_idle_task(p));
                mark_start = wallclock - irqtime;

                /* Roll window over. If IRQ busy time was just in the current
                 * window then that is all that need be accounted. */
                rq->prev_runnable_sum = rq->curr_runnable_sum;
                if (mark_start > window_start) {
                        rq->curr_runnable_sum = scale_exec_time(irqtime, rq);
                        return;
                }

                /* The IRQ busy time spanned multiple windows. Process the
                 * busy time preceding the current window start first. */
                delta = window_start - mark_start;
```

```
                if (delta > window_size)
                        delta = window_size;
                delta = scale_exec_time(delta, rq);
                rq->prev_runnable_sum += delta;

                /* Process the remaining IRQ busy time in the current window. */
                delta = wallclock - window_start;
                rq->curr_runnable_sum = scale_exec_time(delta, rq);

                return;
        }

        BUG();
}
```

* * *

```
static int account_busy_for_task_demand(struct task_struct *p, int event)
{
        /* No need to bother updating task demand for exiting tasks
         * or the idle task. */
        if (exiting_task(p) || is_idle_task(p))
                return 0;

        /* When a task is waking up it is completing a segment of non-busy
         * time. Likewise, if wait time is not treated as busy time, then
         * when a task begins to run or is migrated, it is not running and
         * is completing a segment of non-busy time. */
        if (event == TASK_WAKE || (!walt_account_wait_time &&
                         (event == PICK_NEXT_TASK || event == TASK_MIGRATE)))
                return 0;

        return 1;
}
```

* * *

```
/*
 * Called when new window is starting for a task, to record cpu usage over
 * recently concluded window(s). Normally 'samples' should be 1. It can be > 1
 * when, say, a real-time task runs without preemption for several windows at a
 * stretch.
 */
static void update_history(struct rq *rq, struct task_struct *p,
                         u32 runtime, int samples, int event)
{
        u32 *hist = &p->ravg.sum_history[0];
```

```
int ridx, widx;
u32 max = 0, avg, demand;
u64 sum = 0;

/* Ignore windows where task had no activity */
if (!runtime || is_idle_task(p) || exiting_task(p) || !samples)
                goto done;

/* Push new 'runtime' value onto stack */
widx = walt_ravg_hist_size - 1;
ridx = widx - samples;
for (; ridx >= 0; --widx, --ridx) {
        hist[widx] = hist[ridx];
        sum += hist[widx];
        if (hist[widx] > max)
                max = hist[widx];
}

for (widx = 0; widx < samples && widx < walt_ravg_hist_size; widx++) {
        hist[widx] = runtime;
        sum += hist[widx];
        if (hist[widx] > max)
                max = hist[widx];
}

p->ravg.sum = 0;

if (walt_window_stats_policy == WINDOW_STATS_RECENT) {
        demand = runtime;
} else if (walt_window_stats_policy == WINDOW_STATS_MAX) {
        demand = max;
} else {
        avg = div64_u64(sum, walt_ravg_hist_size);
        if (walt_window_stats_policy == WINDOW_STATS_AVG)
                demand = avg;
        else
                demand = max(avg, runtime);
}

/*
 * A throttled deadline sched class task gets dequeued without
 * changing p->on_rq. Since the dequeue decrements hmp stats
 * avoid decrementing it here again.
 */
if (task_on_rq_queued(p) && (!task_has_dl_policy(p) ||
                                        !p->dl.dl_throttled))
```

```
		fixup_cumulative_runnable_avg(rq, p, demand);
	p->ravg.demand = demand;

done:
	trace_walt_update_history(rq, p, runtime, samples, event);
	return;
}
```

* * *

```
static void add_to_task_demand(struct rq *rq, struct task_struct *p,
				u64 delta)
{
	delta = scale_exec_time(delta, rq);
	p->ravg.sum += delta;
	if (unlikely(p->ravg.sum > walt_ravg_window))
		p->ravg.sum = walt_ravg_window;
}
```

* * *

```
/*
 * Account cpu demand of task and/or update task's cpu demand history
 *
 * ms = p->ravg.mark_start;
 * wc = wallclock
 * ws = rq->window_start
 *
 * Three possibilities:
 *
 *	a) Task event is contained within one window.
 *		window_start < mark_start < wallclock
 *
 *		ws   ms  wc
 *		|    |   |
 *		V    V   V
 *		|---------------|
 *
 *	In this case, p->ravg.sum is updated *iff* event is appropriate
 *	(ex: event == PUT_PREV_TASK)
 *
 *	b) Task event spans two windows.
 *		mark_start < window_start < wallclock
 *
 *		ms   ws   wc
 *		|    |    |
 *		V    V    V
 *              -----|-------------------
 *
 *      In this case, p->ravg.sum is updated with (ws - ms) *iff* event
 *      is appropriate, then a new window sample is recorded followed
 *      by p->ravg.sum being set to (wc - ws) *iff* event is appropriate.
 *
 *      c) Task event spans more than two windows.
 *
 *              ms ws_tmp                          ws  wc
 *              |  |                               |   |
 *              V  V                               V   V
 *              ---|-------|-------|-------|-------|------
 *                 |                               |
 *                 |<------ nr_full_windows ------>|
 *
 *      In this case, p->ravg.sum is updated with (ws_tmp - ms) first *iff*
 *      event is appropriate, window sample of p->ravg.sum is recorded,
 *      'nr_full_window' samples of window_size is also recorded *iff*
 *      event is appropriate and finally p->ravg.sum is set to (wc - ws)
 *      *iff* event is appropriate.
 *
 * IMPORTANT : Leave p->ravg.mark_start unchanged, as update_cpu_busy_time()
 * depends on it!
 */
static void update_task_demand(struct task_struct *p, struct rq *rq,
             int event, u64 wallclock)
{
        u64 mark_start = p->ravg.mark_start;
        u64 delta, window_start = rq->window_start;
        int new_window, nr_full_windows;
        u32 window_size = walt_ravg_window;

        new_window = mark_start < window_start;
        if (!account_busy_for_task_demand(p, event)) {
                if (new_window)
                        /* If the time accounted isn't being accounted as
                         * busy time, and a new window started, only the
                         * previous window need be closed out with the
                         * pre-existing demand. Multiple windows may have
                         * elapsed, but since empty windows are dropped,
                         * it is not necessary to account those. */
                        update_history(rq, p, p->ravg.sum, 1, event);
                return;
        }

        if (!new_window) {
                /* The simple case - busy time contained within the existing
                 * window. */
                add_to_task_demand(rq, p, wallclock - mark_start);
                return;
        }

        /* Busy time spans at least two windows. Temporarily rewind
         * window_start to first window boundary after mark_start. */
        delta = window_start - mark_start;
        nr_full_windows = div64_u64(delta, window_size);
        window_start -= (u64)nr_full_windows * (u64)window_size;

        /* Process (window_start - mark_start) first */
        add_to_task_demand(rq, p, window_start - mark_start);

        /* Push new sample(s) into task's demand history */
        update_history(rq, p, p->ravg.sum, 1, event);
        if (nr_full_windows)
                update_history(rq, p, scale_exec_time(window_size, rq),
                               nr_full_windows, event);

        /* Roll window_start back to current to process any remainder
         * in current window. */
        window_start += (u64)nr_full_windows * (u64)window_size;

        /* Process (wallclock - window_start) next */
        mark_start = window_start;
        add_to_task_demand(rq, p, wallclock - mark_start);
}
```

* * *

```
/* Reflect task activity on its demand and cpu's busy time statistics */
void walt_update_task_ravg(struct task_struct *p, struct rq *rq,
             int event, u64 wallclock, u64 irqtime)
{
        if (walt_disabled || !rq->window_start)
                return;

        lockdep_assert_held(&rq->lock);

        update_window_start(rq, wallclock);

        if (!p->ravg.mark_start)
                goto done;
        update_task_demand(p, rq, event, wallclock);
        update_cpu_busy_time(p, rq, event, wallclock, irqtime);

done:
        trace_walt_update_task_ravg(p, rq, event, wallclock, irqtime);

        p->ravg.mark_start = wallclock;
}
```

* * *

```
unsigned long __weak arch_get_cpu_efficiency(int cpu)
{
        return SCHED_CAPACITY_SCALE;
}
```

* * *

```
void walt_init_cpu_efficiency(void)
{
        int i, efficiency;
        unsigned int max = 0, min = UINT_MAX;

        for_each_possible_cpu(i) {
                efficiency = arch_get_cpu_efficiency(i);
                cpu_rq(i)->efficiency = efficiency;

                if (efficiency > max)
                        max = efficiency;
                if (efficiency < min)
                        min = efficiency;
        }

        if (max)
                max_possible_efficiency = max;

        if (min)
                min_possible_efficiency = min;
}
```

* * *

```
static void reset_task_stats(struct task_struct *p)
{
        u32 sum = 0;

        if (exiting_task(p))
```

```
                sum = EXITING_TASK_MARKER;

        memset(&p->ravg, 0, sizeof(struct ravg));
        /* Retain EXITING_TASK marker */
        p->ravg.sum_history[0] = sum;
}
```

* * *

```
void walt_mark_task_starting(struct task_struct *p)
{
        u64 wallclock;
        struct rq *rq = task_rq(p);

        if (!rq->window_start) {
                reset_task_stats(p);
                return;
        }

        wallclock = walt_ktime_clock();
        p->ravg.mark_start = wallclock;
}
```

* * *

```
void walt_set_window_start(struct rq *rq)
{
        int cpu = cpu_of(rq);
        struct rq *sync_rq = cpu_rq(sync_cpu);

        if (rq->window_start)
                return;

        if (cpu == sync_cpu) {
                rq->window_start = walt_ktime_clock();
        } else {
                raw_spin_unlock(&rq->lock);
                double_rq_lock(rq, sync_rq);
                rq->window_start = cpu_rq(sync_cpu)->window_start;
                rq->curr_runnable_sum = rq->prev_runnable_sum = 0;
                raw_spin_unlock(&sync_rq->lock);
        }

        rq->curr->ravg.mark_start = rq->window_start;
}
```

* * *

```
void walt_migrate_sync_cpu(int cpu)
{
        if (cpu == sync_cpu)
                sync_cpu = smp_processor_id();
}
```

* * *

```
void walt_fixup_busy_time(struct task_struct *p, int new_cpu)
{
        struct rq *src_rq = task_rq(p);
        struct rq *dest_rq = cpu_rq(new_cpu);
        u64 wallclock;

        if (!p->on_rq && p->state != TASK_WAKING)
                return;

        if (exiting_task(p)) {
                return;
        }

        if (p->state == TASK_WAKING)
                double_rq_lock(src_rq, dest_rq);

        wallclock = walt_ktime_clock();

        walt_update_task_ravg(task_rq(p)->curr, task_rq(p),
                        TASK_UPDATE, wallclock, 0);
        walt_update_task_ravg(dest_rq->curr, dest_rq,
                        TASK_UPDATE, wallclock, 0);

        walt_update_task_ravg(p, task_rq(p), TASK_MIGRATE, wallclock, 0);

        if (p->ravg.curr_window) {
                src_rq->curr_runnable_sum -= p->ravg.curr_window;
                dest_rq->curr_runnable_sum += p->ravg.curr_window;
        }

        if (p->ravg.prev_window) {
                src_rq->prev_runnable_sum -= p->ravg.prev_window;
                dest_rq->prev_runnable_sum += p->ravg.prev_window;
        }

        if ((s64)src_rq->prev_runnable_sum < 0) {
                src_rq->prev_runnable_sum = 0;
                WARN_ON(1);
        }
        if ((s64)src_rq->curr_runnable_sum < 0) {
                src_rq->curr_runnable_sum = 0;
                WARN_ON(1);
        }

        trace_walt_migration_update_sum(src_rq, p);
        trace_walt_migration_update_sum(dest_rq, p);

        if (p->state == TASK_WAKING)
                double_rq_unlock(src_rq, dest_rq);
}
```

* * *

```
/*
 * Return 'capacity' of a cpu in reference to "least" efficient cpu, such that
 * least efficient cpu gets capacity of 1024
 */
static unsigned long capacity_scale_cpu_efficiency(int cpu)
{
        return (1024 * cpu_rq(cpu)->efficiency) / min_possible_efficiency;
}
```

* * *

```
/*
 * Return 'capacity' of a cpu in reference to cpu with lowest max_freq
 * (min_max_freq), such that one with lowest max_freq gets capacity of 1024.
 */
static unsigned long capacity_scale_cpu_freq(int cpu)
{
        return (1024 * cpu_rq(cpu)->max_freq) / min_max_freq;
}
```

* * *

```
/*
 * Return load_scale_factor of a cpu in reference to "most" efficient cpu, so
 * that "most" efficient cpu gets a load_scale_factor of 1
 */
static unsigned long load_scale_cpu_efficiency(int cpu)
{
        return DIV_ROUND_UP(1024 * max_possible_efficiency,
                            cpu_rq(cpu)->efficiency);
}
```

* * *

```
/*
 * Return load_scale_factor of a cpu in reference to cpu with best max_freq
 * (max_possible_freq), so that one with best max_freq gets a load_scale_factor
 * of 1.
 */
static unsigned long load_scale_cpu_freq(int cpu)
{
        return DIV_ROUND_UP(1024 * max_possible_freq, cpu_rq(cpu)->max_freq);
}
```

* * *

```
static int compute_capacity(int cpu)
{
        int capacity = 1024;

        capacity *= capacity_scale_cpu_efficiency(cpu);
        capacity >>= 10;

        capacity *= capacity_scale_cpu_freq(cpu);
        capacity >>= 10;

        return capacity;
}
```

* * *

```
static int compute_load_scale_factor(int cpu)
{
        int load_scale = 1024;

        /*
         * load_scale_factor accounts for the fact that task load
         * is in reference to "best" performing cpu. Task's load will need to be
         * scaled (up) by a factor to determine suitability to be placed on a
         * (little) cpu.
         */
        load_scale *= load_scale_cpu_efficiency(cpu);
        load_scale >>= 10;

        load_scale *= load_scale_cpu_freq(cpu);
        load_scale >>= 10;

        return load_scale;
}
```

* * *

```
static int cpufreq_notifier_policy(struct notifier_block *nb,
                unsigned long val, void *data)
{
        struct cpufreq_policy *policy = (struct cpufreq_policy *)data;
        int i, update_max = 0;
        u64 highest_mpc = 0, highest_mplsf = 0;
        const struct cpumask *cpus = policy->related_cpus;
        unsigned int orig_min_max_freq = min_max_freq;
        unsigned int orig_max_possible_freq = max_possible_freq;
        /* Initialized to policy->max in case policy->related_cpus is empty! */
        unsigned int orig_max_freq = policy->max;

        if (val != CPUFREQ_NOTIFY)
                return 0;

        for_each_cpu(i, policy->related_cpus) {
                cpumask_copy(&cpu_rq(i)->freq_domain_cpumask,
                             policy->related_cpus);
                orig_max_freq = cpu_rq(i)->max_freq;
                cpu_rq(i)->min_freq = policy->min;
                cpu_rq(i)->max_freq = policy->max;
                cpu_rq(i)->cur_freq = policy->cur;
                cpu_rq(i)->max_possible_freq = policy->cpuinfo.max_freq;
        }

        max_possible_freq = max(max_possible_freq, policy->cpuinfo.max_freq);
        if (min_max_freq == 1)
                min_max_freq = UINT_MAX;
        min_max_freq = min(min_max_freq, policy->cpuinfo.max_freq);
        BUG_ON(!min_max_freq);
        BUG_ON(!policy->max);

        /* Changes to policy other than max_freq don't require any updates */
        if (orig_max_freq == policy->max)
                return 0;

        /*
         * A changed min_max_freq or max_possible_freq (possible during bootup)
         * needs to trigger re-computation of load_scale_factor and capacity for
         * all possible cpus (even those offline). It also needs to trigger
         * re-computation of nr_big_task count on all online cpus.
         *
         * A changed rq->max_freq otoh needs to trigger re-computation of
         * load_scale_factor and capacity for just the cluster of cpus involved.
         * Since small task definition depends on max_load_scale_factor, a
         * changed load_scale_factor of one cluster could influence
```

```
 * classification of tasks in another cluster. Hence a changed
 * rq->max_freq will need to trigger re-computation of nr_big_task
 * count on all online cpus.
 *
 * While it should be sufficient for nr_big_tasks to be
 * re-computed for only online cpus, we have inadequate context
 * information here (in policy notifier) with regard to hotplug-safety
 * context in which notification is issued. As a result, we can't use
 * get_online_cpus() here, as it can lead to deadlock. Until cpufreq is
 * fixed up to issue notification always in hotplug-safe context,
 * re-compute nr_big_task for all possible cpus.
 */

if (orig_min_max_freq != min_max_freq ||
        orig_max_possible_freq != max_possible_freq) {
                cpus = cpu_possible_mask;
                update_max = 1;
}
/*
 * Changed load_scale_factor can trigger reclassification of tasks as
 * big or small. Make this change "atomic" so that tasks are accounted
 * properly due to changed load_scale_factor
 */
for_each_cpu(i, cpus) {
        struct rq *rq = cpu_rq(i);

        rq->capacity = compute_capacity(i);
        rq->load_scale_factor = compute_load_scale_factor(i);

        if (update_max) {
                u64 mpc, mplsf;

                mpc = div_u64(((u64) rq->capacity) *
                        rq->max_possible_freq, rq->max_freq);
                rq->max_possible_capacity = (int) mpc;

                mplsf = div_u64(((u64) rq->load_scale_factor) *
                        rq->max_possible_freq, rq->max_freq);

                if (mpc > highest_mpc) {
                        highest_mpc = mpc;
                        cpumask_clear(&mpc_mask);
                        cpumask_set_cpu(i, &mpc_mask);
                } else if (mpc == highest_mpc) {
                        cpumask_set_cpu(i, &mpc_mask);
                }
```

```
                        if (mplsf > highest_mplsf)
                                highest_mplsf = mplsf;
                }
        }

        if (update_max) {
                max_possible_capacity = highest_mpc;
                max_load_scale_factor = highest_mplsf;
        }

        return 0;
}
```

* * *

```
static int cpufreq_notifier_trans(struct notifier_block *nb,
                unsigned long val, void *data)
{
        struct cpufreq_freqs *freq = (struct cpufreq_freqs *)data;
        unsigned int cpu = freq->cpu, new_freq = freq->new;
        unsigned long flags;
        int i;

        if (val != CPUFREQ_POSTCHANGE)
                return 0;

        BUG_ON(!new_freq);

        if (cpu_rq(cpu)->cur_freq == new_freq)
                return 0;

        for_each_cpu(i, &cpu_rq(cpu)->freq_domain_cpumask) {
                struct rq *rq = cpu_rq(i);

                raw_spin_lock_irqsave(&rq->lock, flags);
                walt_update_task_ravg(rq->curr, rq, TASK_UPDATE,
                                      walt_ktime_clock(), 0);
                rq->cur_freq = new_freq;
                raw_spin_unlock_irqrestore(&rq->lock, flags);
        }

        return 0;
}
```

* * *

```
static int register_sched_callback(void)
{
        int ret;

        ret = cpufreq_register_notifier(&notifier_policy_block,
                                                CPUFREQ_POLICY_NOTIFIER);

        if (!ret)
                ret = cpufreq_register_notifier(&notifier_trans_block,
                                                CPUFREQ_TRANSITION_NOTIFIER);

        return 0;
}
```

* * *

```
void walt_init_new_task_load(struct task_struct *p)
{
        int i;
        u32 init_load_windows =
                        div64_u64((u64)sysctl_sched_walt_init_task_load_pct *
                          (u64)walt_ravg_window, 100);
        u32 init_load_pct = current->init_load_pct;

        p->init_load_pct = 0;
        memset(&p->ravg, 0, sizeof(struct ravg));

        if (init_load_pct) {
                init_load_windows = div64_u64((u64)init_load_pct *
                          (u64)walt_ravg_window, 100);
        }

        p->ravg.demand = init_load_windows;
        for (i = 0; i < RAVG_HIST_SIZE_MAX; ++i)
                p->ravg.sum_history[i] = init_load_windows;
}
```

6.3 tune.c

```
static int
__schedtune_accept_deltas(int nrg_delta, int cap_delta,
                          int perf_boost_idx, int perf_constrain_idx)
{
        int payoff = -INT_MAX;
        int gain_idx = -1;
        /* Performance Boost (B) region */
        if (nrg_delta >= 0 && cap_delta > 0)
                gain_idx = perf_boost_idx;
        /* Performance Constraint (C) region */
        else if (nrg_delta < 0 && cap_delta <= 0)
                gain_idx = perf_constrain_idx;

        /* Default: reject schedule candidate */
        if (gain_idx == -1)
                return payoff;

        /*
         * Evaluate "Performance Boost" vs "Energy Increase"
         *
         * - Performance Boost (B) region
         *
         *   Condition: nrg_delta > 0 && cap_delta > 0
         *   Payoff criteria:
         *     cap_gain / nrg_gain  < cap_delta / nrg_delta =
         *     cap_gain * nrg_delta < cap_delta * nrg_gain
         *   Note that since both nrg_gain and nrg_delta are positive, the
         *   inequality does not change. Thus:
         *
         *     payoff = (cap_delta * nrg_gain) - (cap_gain * nrg_delta)
         *
         * - Performance Constraint (C) region
         *
         *   Condition: nrg_delta < 0 && cap_delta < 0
         *   payoff criteria:
         *     cap_gain / nrg_gain  > cap_delta / nrg_delta =
         *     cap_gain * nrg_delta < cap_delta * nrg_gain
         *   Note that since nrg_gain > 0 while nrg_delta < 0, the
         *   inequality change. Thus:
         *
         *     payoff = (cap_delta * nrg_gain) - (cap_gain * nrg_delta)
         *
         * This means that, in case of same positive defined {cap,nrg}_gain
         * for both the B and C regions, we can use the same payoff formula
         * where a positive value represents the accept condition.
         */
        payoff  = cap_delta * threshold_gains[gain_idx].nrg_gain;
        payoff -= nrg_delta * threshold_gains[gain_idx].cap_gain;

        return payoff;
}
```

* * *

```
int
schedtune_accept_deltas(int nrg_delta, int cap_delta,
                        struct task_struct *task)
{
        struct schedtune *ct;
        int perf_boost_idx;
        int perf_constrain_idx;

        /* Optimal (O) region */
        if (nrg_delta < 0 && cap_delta > 0) {
                trace_sched_tune_filter(nrg_delta, cap_delta, 0, 0, 1, 0);
                return INT_MAX;
        }

        /* Suboptimal (S) region */
        if (nrg_delta > 0 && cap_delta < 0) {
                trace_sched_tune_filter(nrg_delta, cap_delta, 0, 0, -1, 5);
                return -INT_MAX;
        }

        /* Get task specific perf Boost/Constraints indexes */
        rcu_read_lock();
        ct = task_schedtune(task);
        perf_boost_idx = ct->perf_boost_idx;
        perf_constrain_idx = ct->perf_constrain_idx;
        rcu_read_unlock();

        return __schedtune_accept_deltas(nrg_delta, cap_delta,
                        perf_boost_idx, perf_constrain_idx);
}
```

* * *

```
static void
schedtune_cpu_update(int cpu)
{
        struct boost_groups *bg;
        int boost_max;
        int idx;

        bg = &per_cpu(cpu_boost_groups, cpu);

        /* The root boost group is always active */
        boost_max = bg->group[0].boost;
        for (idx = 1; idx < BOOSTGROUPS_COUNT; ++idx) {
```

```
			/*
			 * A boost group affects a CPU only if it has
			 * RUNNABLE tasks on that CPU
			 */
			if (bg->group[idx].tasks == 0)
				continue;

			boost_max = max(boost_max, bg->group[idx].boost);
		}
		/* Ensures boost_max is non-negative when all cgroup boost values
		 * are neagtive. Avoids under-accounting of cpu capacity which may cause
		 * task stacking and frequency spikes.*/
		boost_max = max(boost_max, 0);
		bg->boost_max = boost_max;
}
```

* * *

```
static int
schedtune_boostgroup_update(int idx, int boost)
{
	struct boost_groups *bg;
	int cur_boost_max;
	int old_boost;
	int cpu;

	/* Update per CPU boost groups */
	for_each_possible_cpu(cpu) {
		bg = &per_cpu(cpu_boost_groups, cpu);

		/*
		 * Keep track of current boost values to compute the per CPU
		 * maximum only when it has been affected by the new value of
		 * the updated boost group
		 */
		cur_boost_max = bg->boost_max;
		old_boost = bg->group[idx].boost;

		/* Update the boost value of this boost group */
		bg->group[idx].boost = boost;

		/* Check if this update increase current max */
		if (boost > cur_boost_max && bg->group[idx].tasks) {
			bg->boost_max = boost;
			trace_sched_tune_boostgroup_update(cpu, 1, bg->boost_max);
			continue;
```

```
                }

                /* Check if this update has decreased current max */
                if (cur_boost_max == old_boost && old_boost > boost) {
                        schedtune_cpu_update(cpu);
                        trace_sched_tune_boostgroup_update(cpu, -1, bg->boost_max);
                        continue;
                }

                trace_sched_tune_boostgroup_update(cpu, 0, bg->boost_max);
        }

        return 0;
}
```

* * *

```
static inline void
schedtune_tasks_update(struct task_struct *p, int cpu, int idx, int task_count)
{
        struct boost_groups *bg = &per_cpu(cpu_boost_groups, cpu);
        int tasks = bg->group[idx].tasks + task_count;

        /* Update boosted tasks count while avoiding to make it negative */
        bg->group[idx].tasks = max(0, tasks);

        trace_sched_tune_tasks_update(p, cpu, tasks, idx,
                        bg->group[idx].boost, bg->boost_max);

        /* Boost group activation or deactivation on that RQ */
        if (tasks == 1 || tasks == 0)
                schedtune_cpu_update(cpu);
}
```

* * *

```
/*
 * NOTE: This function must be called while holding the lock on the CPU RQ
 */
void schedtune_enqueue_task(struct task_struct *p, int cpu)
{
        struct boost_groups *bg = &per_cpu(cpu_boost_groups, cpu);
        unsigned long irq_flags;
        struct schedtune *st;
        int idx;

        if (!unlikely(schedtune_initialized))
```

```
                return;

        /*
         * When a task is marked PF_EXITING by do_exit() it's going to be
         * dequeued and enqueued multiple times in the exit path.
         * Thus we avoid any further update, since we do not want to change
         * CPU boosting while the task is exiting.
         */
        if (p->flags & PF_EXITING)
                return;

        /*
         * Boost group accouting is protected by a per-cpu lock and requires
         * interrupt to be disabled to avoid race conditions for example on
         * do_exit()::cgroup_exit() and task migration.
         */
        raw_spin_lock_irqsave(&bg->lock, irq_flags);
        rcu_read_lock();

        st = task_schedtune(p);
        idx = st->idx;

        schedtune_tasks_update(p, cpu, idx, ENQUEUE_TASK);

        rcu_read_unlock();
        raw_spin_unlock_irqrestore(&bg->lock, irq_flags);
}
```

* * *

```
int schedtune_can_attach(struct cgroup_taskset *tset)
{
        struct task_struct *task;
        struct cgroup_subsys_state *css;
        struct boost_groups *bg;
        struct rq_flags irq_flags;
        unsigned int cpu;
        struct rq *rq;
        int src_bg; /* Source boost group index */
        int dst_bg; /* Destination boost group index */
        int tasks;

        if (!unlikely(schedtune_initialized))
                return 0;
```

```
cgroup_taskset_for_each(task, css, tset) {

        /*
         * Lock the CPU's RQ the task is enqueued to avoid race
         * conditions with migration code while the task is being
         * accounted
         */
        rq = lock_rq_of(task, &irq_flags);

        if (!task->on_rq) {
                unlock_rq_of(rq, task, &irq_flags);
                continue;
        }

        /*
         * Boost group accouting is protected by a per-cpu lock and requires
         * interrupt to be disabled to avoid race conditions on...
         */
        cpu = cpu_of(rq);
        bg = &per_cpu(cpu_boost_groups, cpu);
        raw_spin_lock(&bg->lock);

        dst_bg = css_st(css)->idx;
        src_bg = task_schedtune(task)->idx;

        /*
         * Current task is not changing boostgroup, which can
         * happen when the new hierarchy is in use.
         */
        if (unlikely(dst_bg == src_bg)) {
                raw_spin_unlock(&bg->lock);
                unlock_rq_of(rq, task, &irq_flags);
                continue;
        }

        /*
         * This is the case of a RUNNABLE task which is switching its
         * current boost group.
         */

        /* Move task from src to dst boost group */
        tasks = bg->group[src_bg].tasks - 1;
        bg->group[src_bg].tasks = max(0, tasks);
        bg->group[dst_bg].tasks += 1;

        raw_spin_unlock(&bg->lock);
```

```
                unlock_rq_of(rq, task, &irq_flags);

                /* Update CPU boost group */
                if (bg->group[src_bg].tasks == 0 || bg->group[dst_bg].tasks == 1)
                        schedtune_cpu_update(task_cpu(task));

        }

        return 0;
}
```

* * *

```
void schedtune_cancel_attach(struct cgroup_taskset *tset)
{
        /* This can happen only if SchedTune controller is mounted with
         * other hierarchies ane one of them fails. Since usually SchedTune is
         * mouted on its own hierarcy, for the time being we do not implement
         * a proper rollback mechanism */
        WARN(1, "SchedTune cancel attach not implemented");
}
```

* * *

```
/*
 * NOTE: This function must be called while holding the lock on the CPU RQ
 */
void schedtune_dequeue_task(struct task_struct *p, int cpu)
{
        struct boost_groups *bg = &per_cpu(cpu_boost_groups, cpu);
        unsigned long irq_flags;
        struct schedtune *st;
        int idx;

        if (!unlikely(schedtune_initialized))
                return;

        /*
         * When a task is marked PF_EXITING by do_exit() it's going to be
         * dequeued and enqueued multiple times in the exit path.
         * Thus we avoid any further update, since we do not want to change
         * CPU boosting while the task is exiting.
         * The last dequeue is already enforce by the do_exit() code path
         * via schedtune_exit_task().
         */
        if (p->flags & PF_EXITING)
                return;
	/*
	 * Boost group accouting is protected by a per-cpu lock and requires
	 * interrupt to be disabled to avoid race conditions on...
	 */
	raw_spin_lock_irqsave(&bg->lock, irq_flags);
	rcu_read_lock();

	st = task_schedtune(p);
	idx = st->idx;

	schedtune_tasks_update(p, cpu, idx, DEQUEUE_TASK);

	rcu_read_unlock();
	raw_spin_unlock_irqrestore(&bg->lock, irq_flags);
}
```

* * *

```
void schedtune_exit_task(struct task_struct *tsk)
{
	struct schedtune *st;
	struct rq_flags irq_flags;
	unsigned int cpu;
	struct rq *rq;
	int idx;

	if (!unlikely(schedtune_initialized))
		return;

	rq = lock_rq_of(tsk, &irq_flags);
	rcu_read_lock();

	cpu = cpu_of(rq);
	st = task_schedtune(tsk);
	idx = st->idx;
	schedtune_tasks_update(tsk, cpu, idx, DEQUEUE_TASK);

	rcu_read_unlock();
	unlock_rq_of(rq, tsk, &irq_flags);
}
```

* * *

```
int schedtune_cpu_boost(int cpu)
{
	struct boost_groups *bg;
```

```
        bg = &per_cpu(cpu_boost_groups, cpu);
        return bg->boost_max;
}
```

* * *

```
int schedtune_task_boost(struct task_struct *p)
{
        struct schedtune *st;
        int task_boost;

        /* Get task boost value */
        rcu_read_lock();
        st = task_schedtune(p);
        task_boost = st->boost;
        rcu_read_unlock();

        return task_boost;
}
```

* * *

```
int schedtune_prefer_idle(struct task_struct *p)
{
        struct schedtune *st;
        int prefer_idle;

        /* Get prefer_idle value */
        rcu_read_lock();
        st = task_schedtune(p);
        prefer_idle = st->prefer_idle;
        rcu_read_unlock();

        return prefer_idle;
}
```

* * *

```
static u64
prefer_idle_read(struct cgroup_subsys_state *css, struct cftype *cft)
{
        struct schedtune *st = css_st(css);

        return st->prefer_idle;
}
```

* * *

```
static int
prefer_idle_write(struct cgroup_subsys_state *css, struct cftype *cft,
            u64 prefer_idle)
{
        struct schedtune *st = css_st(css);
        st->prefer_idle = prefer_idle;

        return 0;
}
```

* * *

```
static s64
boost_read(struct cgroup_subsys_state *css, struct cftype *cft)
{
        struct schedtune *st = css_st(css);

        return st->boost;
}
```

* * *

```
static int
boost_write(struct cgroup_subsys_state *css, struct cftype *cft,
            s64 boost)
{
        struct schedtune *st = css_st(css);
        unsigned threshold_idx;
        int boost_pct;

        if (boost < -100 || boost > 100)
                return -EINVAL;
        boost_pct = boost;

        /*
         * Update threshold params for Performance Boost (B)
         * and Performance Constraint (C) regions.
         * The current implementatio uses the same cuts for both
         * B and C regions.
         */
        threshold_idx = clamp(boost_pct, 0, 99) / 10;
        st->perf_boost_idx = threshold_idx;
        st->perf_constrain_idx = threshold_idx;

        st->boost = boost;
        if (css == &root_schedtune.css) {
                sysctl_sched_cfs_boost = boost;
                perf_boost_idx  = threshold_idx;
                perf_constrain_idx  = threshold_idx;
        }

        /* Update CPU boost */
        schedtune_boostgroup_update(st->idx, st->boost);

        trace_sched_tune_config(st->boost);

        return 0;
}
```

* * *

```
static int
schedtune_boostgroup_init(struct schedtune *st)
{
        struct boost_groups *bg;
        int cpu;

        /* Keep track of allocated boost groups */
        allocated_group[st->idx] = st;

        /* Initialize the per CPU boost groups */
        for_each_possible_cpu(cpu) {
                bg = &per_cpu(cpu_boost_groups, cpu);
                bg->group[st->idx].boost = 0;
                bg->group[st->idx].tasks = 0;
        }

        return 0;
}
```

* * *

```
static struct cgroup_subsys_state *
schedtune_css_alloc(struct cgroup_subsys_state *parent_css)
{
        struct schedtune *st;
        int idx;

        if (!parent_css)
                return &root_schedtune.css;

        /* Allow only single level hierachies */
        if (parent_css != &root_schedtune.css) {
                pr_err("Nested SchedTune boosting groups not allowed\n");
                return ERR_PTR(-ENOMEM);
        }

        /* Allow only a limited number of boosting groups */
        for (idx = 1; idx < BOOSTGROUPS_COUNT; ++idx)
                if (!allocated_group[idx])
                        break;
        if (idx == BOOSTGROUPS_COUNT) {
                pr_err("Trying to create more than %d SchedTune boosting groups\n",
                       BOOSTGROUPS_COUNT);
                return ERR_PTR(-ENOSPC);
        }

        st = kzalloc(sizeof(*st), GFP_KERNEL);
        if (!st)
                goto out;

        /* Initialize per CPUs boost group support */
        st->idx = idx;
        if (schedtune_boostgroup_init(st))
                goto release;

        return &st->css;

release:
        kfree(st);
out:
        return ERR_PTR(-ENOMEM);
}
```

* * *

```
static void
schedtune_boostgroup_release(struct schedtune *st)
{
        /* Reset this boost group */
        schedtune_boostgroup_update(st->idx, 0);

        /* Keep track of allocated boost groups */
        allocated_group[st->idx] = NULL;
}
```

* * *

```
static void
schedtune_css_free(struct cgroup_subsys_state *css)
{
        struct schedtune *st = css_st(css);

        schedtune_boostgroup_release(st);
        kfree(st);
}
```

* * *

```
static inline void
schedtune_init_cgroups(void)
{
        struct boost_groups *bg;
        int cpu;

        /* Initialize the per CPU boost groups */
        for_each_possible_cpu(cpu) {
                bg = &per_cpu(cpu_boost_groups, cpu);
                memset(bg, 0, sizeof(struct boost_groups));
                raw_spin_lock_init(&bg->lock);
        }

        pr_info("schedtune: configured to support %d boost groups\n",
                BOOSTGROUPS_COUNT);

        schedtune_initialized = true;
}
```

* * *

```
int
schedtune_accept_deltas(int nrg_delta, int cap_delta,
                        struct task_struct *task)
{
        /* Optimal (O) region */
        if (nrg_delta < 0 && cap_delta > 0) {
                trace_sched_tune_filter(nrg_delta, cap_delta, 0, 0, 1, 0);
                return INT_MAX;
        }

        /* Suboptimal (S) region */
        if (nrg_delta > 0 && cap_delta < 0) {
                trace_sched_tune_filter(nrg_delta, cap_delta, 0, 0, -1, 5);
                return -INT_MAX;
        }

        return __schedtune_accept_deltas(nrg_delta, cap_delta,
                        perf_boost_idx, perf_constrain_idx);
}
```

* * *

```
int
sysctl_sched_cfs_boost_handler(struct ctl_table *table, int write,
                               void __user *buffer, size_t *lenp,
                               loff_t *ppos)
{
        int ret = proc_dointvec_minmax(table, write, buffer, lenp, ppos);
        unsigned threshold_idx;
        int boost_pct;

        if (ret || !write)
                return ret;

        if (sysctl_sched_cfs_boost < -100 || sysctl_sched_cfs_boost > 100)
                return -EINVAL;
        boost_pct = sysctl_sched_cfs_boost;

        /*
         * Update threshold params for Performance Boost (B)
         * and Performance Constraint (C) regions.
         * The current implementatio uses the same cuts for both
         * B and C regions.
         */
        threshold_idx = clamp(boost_pct, 0, 99) / 10;
        perf_boost_idx = threshold_idx;
        perf_constrain_idx = threshold_idx;

        return 0;
}
```

* * *

```
static void
schedtune_test_nrg(unsigned long delta_pwr)
{
        unsigned long test_delta_pwr;
        unsigned long test_norm_pwr;
        int idx;

        /*
         * Check normalization constants using some constant system
         * energy values
         */
        pr_info("schedtune: verify normalization constants...\n");
        for (idx = 0; idx < 6; ++idx) {
                test_delta_pwr = delta_pwr >> idx;
                /* Normalize on max energy for target platform */
                test_norm_pwr = reciprocal_divide(
                                        test_delta_pwr << SCHED_CAPACITY_SHIFT,
                                        schedtune_target_nrg.rdiv);

                pr_info("schedtune: max_pwr/2^%d: %4lu => norm_pwr: %5lu\n",
                        idx, test_delta_pwr, test_norm_pwr);
        }
}
```

* * *

```
/*
 * Compute the min/max power consumption of a cluster and all its CPUs
 */
static void
schedtune_add_cluster_nrg(
                struct sched_domain *sd,
                struct sched_group *sg,
                struct target_nrg *ste)
{
        struct sched_domain *sd2;
        struct sched_group *sg2;

        struct cpumask *cluster_cpus;
        char str[32];

        unsigned long min_pwr;
        unsigned long max_pwr;
        int cpu;

        /* Get Cluster energy using EM data for the first CPU */
        cluster_cpus = sched_group_cpus(sg);
        snprintf(str, 32, "CLUSTER[%*pbl]",
                 cpumask_pr_args(cluster_cpus));

        min_pwr = sg->sge->idle_states[sg->sge->nr_idle_states - 1].power;
        max_pwr = sg->sge->cap_states[sg->sge->nr_cap_states - 1].power;
        pr_info("schedtune: %-17s min_pwr: %5lu max_pwr: %5lu\n",
                str, min_pwr, max_pwr);

        /*
         * Keep track of this cluster's energy in the computation of the
         * overall system energy
         */
        ste->min_power += min_pwr;
        ste->max_power += max_pwr;

        /* Get CPU energy using EM data for each CPU in the group */
        for_each_cpu(cpu, cluster_cpus) {
                /* Get a SD view for the specific CPU */
                for_each_domain(cpu, sd2) {
                        /* Get the CPU group */
                        sg2 = sd2->groups;
                        min_pwr = sg2->sge->idle_states[sg2->sge->nr_idle_states - 1].power
                        max_pwr = sg2->sge->cap_states[sg2->sge->nr_cap_states - 1].power;

                        ste->min_power += min_pwr;
                        ste->max_power += max_pwr;

                        snprintf(str, 32, "CPU[%d]", cpu);
                        pr_info("schedtune: %-17s min_pwr: %5lu max_pwr: %5lu\n",
                                str, min_pwr, max_pwr);

                        /*
                         * Assume we have EM data only at the CPU and
                         * the upper CLUSTER level
                         */
                        BUG_ON(!cpumask_equal(
                                sched_group_cpus(sg),
                                sched_group_cpus(sd2->parent->groups)
                                ));
                        break;
                }
        }
}
```

* * *

```
/*
 * Initialize the constants required to compute normalized energy.
 * The values of these constants depends on the EM data for the specific
 * target system and topology.
 * Thus, this function is expected to be called by the code
 * that bind the EM to the topology information.
 */
static int
schedtune_init(void)
{
        struct target_nrg *ste = &schedtune_target_nrg;
        unsigned long delta_pwr = 0;
        struct sched_domain *sd;
        struct sched_group *sg;

        pr_info("schedtune: init normalization constants...\n");
        ste->max_power = 0;
        ste->min_power = 0;

        rcu_read_lock();

        /*
         * When EAS is in use, we always have a pointer to the highest SD
         * which provides EM data.
         */
        sd = rcu_dereference(per_cpu(sd_ea, cpumask_first(cpu_online_mask)));
        if (!sd) {
                pr_info("schedtune: no energy model data\n");
                goto nodata;
        }

        sg = sd->groups;
        do {
                schedtune_add_cluster_nrg(sd, sg, ste);
        } while (sg = sg->next, sg != sd->groups);

        rcu_read_unlock();

        pr_info("schedtune: %-17s min_pwr: %5lu max_pwr: %5lu\n",
                "SYSTEM", ste->min_power, ste->max_power);

        /* Compute normalization constants */
        delta_pwr = ste->max_power - ste->min_power;
        ste->rdiv = reciprocal_value(delta_pwr);
        pr_info("schedtune: using normalization constants mul: %u sh1: %u sh2: %u\n",
                ste->rdiv.m, ste->rdiv.sh1, ste->rdiv.sh2);

        schedtune_test_nrg(delta_pwr);

#ifdef CONFIG_CGROUP_SCHEDTUNE
        schedtune_init_cgroups();
#else
        pr_info("schedtune: configured to support global boosting only\n");
#endif

        return 0;

nodata:
        rcu_read_unlock();
        return -EINVAL;
}
```

6.4 energy.c

```
static void free_resources(void)
{
        int cpu, sd_level;
        struct sched_group_energy *sge;

        for_each_possible_cpu(cpu) {
                for_each_possible_sd_level(sd_level) {
                        sge = sge_array[cpu][sd_level];
                        if (sge) {
                                kfree(sge->cap_states);
                                kfree(sge->idle_states);
                                kfree(sge);
                        }
                }
        }
}
```

* * *

```
void init_sched_energy_costs(void)
{
        struct device_node *cn, *cp;
        struct capacity_state *cap_states;
        struct idle_state *idle_states;
        struct sched_group_energy *sge;
        const struct property *prop;
        int sd_level, i, nstates, cpu;
        const __be32 *val;

        for_each_possible_cpu(cpu) {
                cn = of_get_cpu_node(cpu, NULL);
                if (!cn) {
                        pr_warn("CPU device node missing for CPU %d\n", cpu);
                        return;
                }

                if (!of_find_property(cn, "sched-energy-costs", NULL)) {
                        pr_warn("CPU device node has no sched-energy-costs\n");
                        return;
                }
```

```
for_each_possible_sd_level(sd_level) {
        cp = of_parse_phandle(cn, "sched-energy-costs", sd_level);
        if (!cp)
                break;

        prop = of_find_property(cp, "busy-cost-data", NULL);
        if (!prop || !prop->value) {
                pr_warn("No busy-cost data, skipping sched_energy init\n");
                goto out;
        }

        sge = kcalloc(1, sizeof(struct sched_group_energy),
                      GFP_NOWAIT);

        nstates = (prop->length / sizeof(u32)) / 2;
        cap_states = kcalloc(nstates,
                             sizeof(struct capacity_state),
                             GFP_NOWAIT);

        for (i = 0, val = prop->value; i < nstates; i++) {
                cap_states[i].cap = be32_to_cpup(val++);
                cap_states[i].power = be32_to_cpup(val++);
        }

        sge->nr_cap_states = nstates;
        sge->cap_states = cap_states;

        prop = of_find_property(cp, "idle-cost-data", NULL);
        if (!prop || !prop->value) {
                pr_warn("No idle-cost data, skipping sched_energy init\n");
                goto out;
        }

        nstates = (prop->length / sizeof(u32));
        idle_states = kcalloc(nstates,
                              sizeof(struct idle_state),
                              GFP_NOWAIT);

        for (i = 0, val = prop->value; i < nstates; i++)
                idle_states[i].power = be32_to_cpup(val++);

        sge->nr_idle_states = nstates;
        sge->idle_states = idle_states;

        sge_array[cpu][sd_level] = sge;
```

```
                }
        }

        pr_info("Sched-energy-costs installed from DT\n");
        return;

out:
        free_resources();
}
```

6.5 core.c

```
struct rq *
lock_rq_of(struct task_struct *p, struct rq_flags *flags)
{
        return task_rq_lock(p, flags);
}
```

* * *

```
void
unlock_rq_of(struct rq *rq, struct task_struct *p, struct rq_flags *flags)
{
        task_rq_unlock(rq, p, flags);
}
```

* * *

```
#ifdef CONFIG_CPU_QUIET
u64 nr_running_integral(unsigned int cpu)
{
        unsigned int seqcnt;
        u64 integral;
        struct rq *q;

        if (cpu >= nr_cpu_ids)
                return 0;

        q = cpu_rq(cpu);

        /*
         * Update average to avoid reading stalled value if there were
         * no run-queue changes for a long time. On the other hand if
         * the changes are happening right now, just read current value
         * directly.
         */
        seqcnt = read_seqcount_begin(&q->ave_seqcnt);
        integral = do_nr_running_integral(q);
        if (read_seqcount_retry(&q->ave_seqcnt, seqcnt)) {
                read_seqcount_begin(&q->ave_seqcnt);
                integral = q->nr_running_integral;
        }

        return integral;
}
#endif
```

* * *

```
static inline
unsigned long add_capacity_margin(unsigned long cpu_capacity)
{
        cpu_capacity  = cpu_capacity * capacity_margin;
        cpu_capacity /= SCHED_CAPACITY_SCALE;
        return cpu_capacity;
}
```

* * *

```
static inline
unsigned long sum_capacity_reqs(unsigned long cfs_cap,
                                struct sched_capacity_reqs *scr)
{
        unsigned long total = add_capacity_margin(cfs_cap + scr->rt);
        return total += scr->dl;
}
```

* * *

```
static void sched_freq_tick_pelt(int cpu)
{
        unsigned long cpu_utilization = capacity_max;
        unsigned long capacity_curr = capacity_curr_of(cpu);
        struct sched_capacity_reqs *scr;

        scr = &per_cpu(cpu_sched_capacity_reqs, cpu);
        if (sum_capacity_reqs(cpu_utilization, scr) < capacity_curr)
                return;

        /*
         * To make free room for a task that is building up its "real"
         * utilization and to harm its performance the least, request
         * a jump to a higher OPP as soon as the margin of free capacity
         * is impacted (specified by capacity_margin).
         */
        set_cfs_cpu_capacity(cpu, true, cpu_utilization);
}
```

* * *

```
#ifdef CONFIG_SCHED_WALT
static void sched_freq_tick_walt(int cpu)
{
        unsigned long cpu_utilization = cpu_util(cpu);
        unsigned long capacity_curr = capacity_curr_of(cpu);

        if (walt_disabled || !sysctl_sched_use_walt_cpu_util)
                return sched_freq_tick_pelt(cpu);

        /*
         * Add a margin to the WALT utilization.
         * NOTE: WALT tracks a single CPU signal for all the scheduling
         * classes, thus this margin is going to be added to the DL class as
         * well, which is something we do not do in sched_freq_tick_pelt case.
         */
        cpu_utilization = add_capacity_margin(cpu_utilization);
        if (cpu_utilization <= capacity_curr)
                return;

        /*
         * It is likely that the load is growing so we
         * keep the added margin in our request as an
         * extra boost.
         */
        set_cfs_cpu_capacity(cpu, true, cpu_utilization);

}
```

* * *

```
static void sched_freq_tick(int cpu)
{
        unsigned long capacity_orig, capacity_curr;

        if (!sched_freq())
                return;

        capacity_orig = capacity_orig_of(cpu);
        capacity_curr = capacity_curr_of(cpu);
        if (capacity_curr == capacity_orig)
```

```
		return;

	_sched_freq_tick(cpu);
}
```

* * *

```
/*
 * Check that the per-cpu provided sd energy data is consistent for all cpus
 * within the mask.
 */
static inline void check_sched_energy_data(int cpu, sched_domain_energy_f fn,
					   const struct cpumask *cpumask)
{
	const struct sched_group_energy * const sge = fn(cpu);
	struct cpumask mask;
	int i;

	if (cpumask_weight(cpumask) <= 1)
		return;

	cpumask_xor(&mask, cpumask, get_cpu_mask(cpu));

	for_each_cpu(i, &mask) {
		const struct sched_group_energy * const e = fn(i);
		int y;

		BUG_ON(e->nr_idle_states != sge->nr_idle_states);

		for (y = 0; y < (e->nr_idle_states); y++) {
			BUG_ON(e->idle_states[y].power !=
					sge->idle_states[y].power);
		}

		BUG_ON(e->nr_cap_states != sge->nr_cap_states);

		for (y = 0; y < (e->nr_cap_states); y++) {
			BUG_ON(e->cap_states[y].cap != sge->cap_states[y].cap);
			BUG_ON(e->cap_states[y].power !=
					sge->cap_states[y].power);
		}
	}
}
```

* * *

```
static void init_sched_energy(int cpu, struct sched_domain *sd,
			      sched_domain_energy_f fn)
```

```
{
        if (!(fn && fn(cpu)))
                return;

        if (cpu != group_balance_cpu(sd->groups))
                return;

        if (sd->child && !sd->child->groups->sge) {
                pr_err("BUG: EAS setup broken for CPU%d\n", cpu);
#ifdef CONFIG_SCHED_DEBUG
                pr_err("     energy data on %s but not on %s domain\n",
                        sd->name, sd->child->name);
#endif
                return;
        }

        check_sched_energy_data(cpu, fn, sched_group_cpus(sd->groups));

        sd->groups->sge = fn(cpu);
}
```

6.6 sched.h

```
static inline void idle_set_state_idx(struct rq *rq, int idle_state_idx)
{
        rq->idle_state_idx = idle_state_idx;
}
```

* * *

```
static inline int idle_get_state_idx(struct rq *rq)
{
        WARN_ON(!rcu_read_lock_held());
        return rq->idle_state_idx;
}
```

* * *

```
static inline u64 do_nr_running_integral(struct rq *rq)
{
        s64 nr, deltax;
        u64 nr_running_integral = rq->nr_running_integral;

        deltax = rq->clock_task - rq->nr_last_stamp;
        nr = NR_AVE_SCALE(rq->nr_running);

        nr_running_integral += nr * deltax;
        return nr_running_integral;
}
```

* * *

```
static inline void add_nr_running(struct rq *rq, unsigned count)
{
        write_seqcount_begin(&rq->ave_seqcnt);
        rq->nr_running_integral = do_nr_running_integral(rq);
        rq->nr_last_stamp = rq->clock_task;
        __add_nr_running(rq, count);
        write_seqcount_end(&rq->ave_seqcnt);
}
```

* * *

```
static inline void sub_nr_running(struct rq *rq, unsigned count)
{
        write_seqcount_begin(&rq->ave_seqcnt);
        rq->nr_running_integral = do_nr_running_integral(rq);
        rq->nr_last_stamp = rq->clock_task;
        __sub_nr_running(rq, count);
        write_seqcount_end(&rq->ave_seqcnt);
}
```

* * *

```
static inline unsigned long capacity_of(int cpu)
{
        return cpu_rq(cpu)->cpu_capacity;
}
```

* * *

```
static inline unsigned long capacity_orig_of(int cpu)
{
        return cpu_rq(cpu)->cpu_capacity_orig;
}
```

* * *

```
/*
 * cpu_util returns the amount of capacity of a CPU that is used by CFS
 * tasks. The unit of the return value must be the one of capacity so we can
 * compare the utilization with the capacity of the CPU that is available for
 * CFS task (ie cpu_capacity).
 *
 * cfs_rq.avg.util_avg is the sum of running time of runnable tasks plus the
 * recent utilization of currently non-runnable tasks on a CPU. It represents
 * the amount of utilization of a CPU in the range [0..capacity_orig] where
 * capacity_orig is the cpu_capacity available at the highest frequency
 * (arch_scale_freq_capacity()).
 * The utilization of a CPU converges towards a sum equal to or less than the
 * current capacity (capacity_curr <= capacity_orig) of the CPU because it is
 * the running time on this CPU scaled by capacity_curr.
 *
 * Nevertheless, cfs_rq.avg.util_avg can be higher than capacity_curr or even
 * higher than capacity_orig because of unfortunate rounding in
 * cfs.avg.util_avg or just after migrating tasks and new task wakeups until
 * the average stabilizes with the new running time. We need to check that the
 * utilization stays within the range of [0..capacity_orig] and cap it if
 * necessary. Without utilization capping, a group could be seen as overloaded
 * (CPU0 utilization at 121\% + CPU1 utilization at 80\%) whereas CPU1 has 20\% of
 * available capacity. We allow utilization to overshoot capacity_curr (but not
 * capacity_orig) as it useful for predicting the capacity required after task
 * migrations (scheduler-driven DVFS).
 */
static inline unsigned long __cpu_util(int cpu, int delta)
{
        unsigned long util = cpu_rq(cpu)->cfs.avg.util_avg;
        unsigned long capacity = capacity_orig_of(cpu);

#ifdef CONFIG_SCHED_WALT
        if (!walt_disabled && sysctl_sched_use_walt_cpu_util) {
                util = cpu_rq(cpu)->prev_runnable_sum << SCHED_CAPACITY_SHIFT;
                util = div_u64(util, walt_ravg_window);
        }
#endif
        delta += util;
        if (delta < 0)
                return 0;

        return (delta >= capacity) ? capacity : delta;
}
```

* * *

```
static inline unsigned long cpu_util(int cpu)
{
        return __cpu_util(cpu, 0);
}
```

* * *

```
static inline bool sched_freq(void)
{
        return static_key_false(&__sched_freq);
}
```

* * *

```
static inline void set_cfs_cpu_capacity(int cpu, bool request,
                                        unsigned long capacity)
{
        struct sched_capacity_reqs *scr = &per_cpu(cpu_sched_capacity_reqs, cpu);

#ifdef CONFIG_SCHED_WALT
       if (!walt_disabled && sysctl_sched_use_walt_cpu_util) {
                int rtdl = scr->rt + scr->dl;
                /*
                 * WALT tracks the utilization of a CPU considering the load
                 * generated by all the scheduling classes.
                 * Since the following call to:
                 *    update_cpu_capacity
                 * is already adding the RT and DL utilizations let's remove
                 * these contributions from the WALT signal.
                 */
                if (capacity > rtdl)
                        capacity -= rtdl;
                else
                        capacity = 0;
        }
#endif
        if (scr->cfs != capacity) {
                scr->cfs = capacity;
                update_cpu_capacity_request(cpu, request);
        }
}
```

* * *

```
static inline void set_rt_cpu_capacity(int cpu, bool request,
                                       unsigned long capacity)
{
        if (per_cpu(cpu_sched_capacity_reqs, cpu).rt != capacity) {
                per_cpu(cpu_sched_capacity_reqs, cpu).rt = capacity;
                update_cpu_capacity_request(cpu, request);
        }
}
```

* * *

```
static inline void set_dl_cpu_capacity(int cpu, bool request,
                                       unsigned long capacity)
{
        if (per_cpu(cpu_sched_capacity_reqs, cpu).dl != capacity) {
                per_cpu(cpu_sched_capacity_reqs, cpu).dl = capacity;
                update_cpu_capacity_request(cpu, request);
        }
}
```

6.7 rt.c

```
static void dump_throttled_rt_tasks(struct rt_rq *rt_rq)
{
        struct rt_prio_array *array = &rt_rq->active;
        struct sched_rt_entity *rt_se;
        char buf[500];
        char *pos = buf;
        char *end = buf + sizeof(buf);
        int idx;

        pos += snprintf(pos, sizeof(buf),
                "sched: RT throttling activated for rt_rq %p (cpu %d)\n",
                rt_rq, cpu_of(rq_of_rt_rq(rt_rq)));

        if (bitmap_empty(array->bitmap, MAX_RT_PRIO))
                goto out;

        pos += snprintf(pos, end - pos, "potential CPU hogs:\n");
        idx = sched_find_first_bit(array->bitmap);
        while (idx < MAX_RT_PRIO) {
                list_for_each_entry(rt_se, array->queue + idx, run_list) {
                        struct task_struct *p;

                        if (!rt_entity_is_task(rt_se))
                                continue;

                        p = rt_task_of(rt_se);
                        if (pos < end)
                                pos += snprintf(pos, end - pos, "\t%s (%d)\n",
                                        p->comm, p->pid);
                }
                idx = find_next_bit(array->bitmap, MAX_RT_PRIO, idx + 1);
        }
out:
#ifdef CONFIG_PANIC_ON_RT_THROTTLING
        /*
```

```
         * Use pr_err() in the BUG() case since printk_sched() will
         * not get flushed and deadlock is not a concern.
         */
        pr_err("%s", buf);
        BUG();
#else
        printk_deferred("%s", buf);
#endif
}
```

6.8 cpufreq_sched.c

```
static void cpufreq_sched_try_driver_target(struct cpufreq_policy *policy,
                                            unsigned int freq)
{
        struct gov_data *gd = policy->governor_data;

        /* avoid race with cpufreq_sched_stop */
        if (!down_write_trylock(&policy->rwsem))
                return;

        __cpufreq_driver_target(policy, freq, CPUFREQ_RELATION_L);

        gd->up_throttle = ktime_add_ns(ktime_get(), gd->up_throttle_nsec);
        gd->down_throttle = ktime_add_ns(ktime_get(), gd->down_throttle_nsec);
        up_write(&policy->rwsem);
}
```

* * *

```
static bool finish_last_request(struct gov_data *gd, unsigned int cur_freq)
{
        ktime_t now = ktime_get();

        ktime_t throttle = gd->requested_freq < cur_freq ?
                gd->down_throttle : gd->up_throttle;

        if (ktime_after(now, throttle))
                return false;

        while (1) {
                int usec_left = ktime_to_ns(ktime_sub(throttle, now));

                usec_left /= NSEC_PER_USEC;
                trace_cpufreq_sched_throttled(usec_left);
                usleep_range(usec_left, usec_left + 100);
                now = ktime_get();
                if (ktime_after(now, throttle))
                        return true;
        }
}
```

* * *

```
/*
 * we pass in struct cpufreq_policy. This is safe because changing out the
 * policy requires a call to __cpufreq_governor(policy, CPUFREQ_GOV_STOP),
 * which tears down all of the data structures and __cpufreq_governor(policy,
 * CPUFREQ_GOV_START) will do a full rebuild, including this kthread with the
 * new policy pointer
 */
static int cpufreq_sched_thread(void *data)
{
        struct sched_param param;
        struct cpufreq_policy *policy;
        struct gov_data *gd;
        unsigned int new_request = 0;
        unsigned int last_request = 0;
        int ret;

        policy = (struct cpufreq_policy *) data;
        gd = policy->governor_data;

        param.sched_priority = 50;
        ret = sched_setscheduler_nocheck(gd->task, SCHED_FIFO, &param);
        if (ret) {
                pr_warn("\%s: failed to set SCHED_FIFO\n", __func__);
                do_exit(-EINVAL);
        } else {
                pr_debug("\%s: kthread (\%d) set to SCHED_FIFO\n",
                                __func__, gd->task->pid);
        }

        do {
                new_request = gd->requested_freq;
                if (new_request == last_request) {
                        set_current_state(TASK_INTERRUPTIBLE);
                        if (kthread_should_stop())
                                break;
                        schedule();
                } else {
                        /*
```

```
                         * if the frequency thread sleeps while waiting to be
                         * unthrottled, start over to check for a newer request
                         */
                        if (finish_last_request(gd, policy->cur))
                                continue;
                        last_request = new_request;
                        cpufreq_sched_try_driver_target(policy, new_request);
                }
        } while (!kthread_should_stop());

        return 0;
}
```

* * *

```
static void cpufreq_sched_irq_work(struct irq_work *irq_work)
{
        struct gov_data *gd;

        gd = container_of(irq_work, struct gov_data, irq_work);
        if (!gd)
                return;

        wake_up_process(gd->task);
}
```

* * *

```
static void update_fdomain_capacity_request(int cpu)
{
        unsigned int freq_new, index_new, cpu_tmp;
        struct cpufreq_policy *policy;
        struct gov_data *gd;
        unsigned long capacity = 0;

        /*
         * Avoid grabbing the policy if possible. A test is still
         * required after locking the CPU's policy to avoid racing
         * with the governor changing.
         */
        if (!per_cpu(enabled, cpu))
                return;

        policy = cpufreq_cpu_get(cpu);
        if (IS_ERR_OR_NULL(policy))
                return;
        if (policy->governor != &cpufreq_gov_sched ||
            !policy->governor_data)
                goto out;

        gd = policy->governor_data;

        /* find max capacity requested by cpus in this policy */
        for_each_cpu(cpu_tmp, policy->cpus) {
                struct sched_capacity_reqs *scr;

                scr = &per_cpu(cpu_sched_capacity_reqs, cpu_tmp);
                capacity = max(capacity, scr->total);
        }

        /* Convert the new maximum capacity request into a cpu frequency */
        freq_new = capacity * policy->max >> SCHED_CAPACITY_SHIFT;
        index_new = cpufreq_frequency_table_target(policy, freq_new, CPUFREQ_RELATION_L);
        freq_new = policy->freq_table[index_new].frequency;

        if (freq_new > policy->max)
                freq_new = policy->max;

        if (freq_new < policy->min)
                freq_new = policy->min;

        trace_cpufreq_sched_request_opp(cpu, capacity, freq_new,
                                        gd->requested_freq);
        if (freq_new == gd->requested_freq)
                goto out;

        gd->requested_freq = freq_new;

        /*
         * Throttling is not yet supported on platforms with fast cpufreq
         * drivers.
         */
        if (cpufreq_driver_slow)
                irq_work_queue_on(&gd->irq_work, cpu);
        else
                cpufreq_sched_try_driver_target(policy, freq_new);

out:
        cpufreq_cpu_put(policy);
}
```

* * *

```
void update_cpu_capacity_request(int cpu, bool request)
{
        unsigned long new_capacity;
        struct sched_capacity_reqs *scr;

        /* The rq lock serializes access to the CPU's sched_capacity_reqs. */
        lockdep_assert_held(&cpu_rq(cpu)->lock);

        scr = &per_cpu(cpu_sched_capacity_reqs, cpu);

        new_capacity = scr->cfs + scr->rt;
        new_capacity = new_capacity * capacity_margin
                / SCHED_CAPACITY_SCALE;
        new_capacity += scr->dl;

        if (new_capacity == scr->total)
                return;

        trace_cpufreq_sched_update_capacity(cpu, request, scr, new_capacity);

        scr->total = new_capacity;
        if (request)
                update_fdomain_capacity_request(cpu);
}
```

* * *

```
static inline void set_sched_freq(void)
{
        static_key_slow_inc(&__sched_freq);
}
```

* * *

```
static inline void clear_sched_freq(void)
{
        static_key_slow_dec(&__sched_freq);
}
```

* * *

```
static int cpufreq_sched_policy_init(struct cpufreq_policy *policy)
{
        struct gov_data *gd;
        int cpu;
        int rc;
        for_each_cpu(cpu, policy->cpus)
                memset(&per_cpu(cpu_sched_capacity_reqs, cpu), 0,
                       sizeof(struct sched_capacity_reqs));

        gd = kzalloc(sizeof(*gd), GFP_KERNEL);
        if (!gd)
                return -ENOMEM;

        gd->up_throttle_nsec = policy->cpuinfo.transition_latency ?
                            policy->cpuinfo.transition_latency :
                            THROTTLE_UP_NSEC;
        gd->down_throttle_nsec = THROTTLE_DOWN_NSEC;
        pr_debug("%s: throttle threshold = %u [ns]\n",
                  __func__, gd->up_throttle_nsec);

        rc = sysfs_create_group(&policy->kobj, get_sysfs_attr());
        if (rc) {
                pr_err("%s: couldn't create sysfs attributes: %d\n", __func__, rc);
                goto err;
        }

        policy->governor_data = gd;
        if (cpufreq_driver_is_slow()) {
                cpufreq_driver_slow = true;
                gd->task = kthread_create(cpufreq_sched_thread, policy,
                                          "kschedfreq:%d",
                                          cpumask_first(policy->related_cpus));
                if (IS_ERR_OR_NULL(gd->task)) {
                        pr_err("%s: failed to create kschedfreq thread\n",
                               __func__);
                        goto err;
                }
                get_task_struct(gd->task);
                kthread_bind_mask(gd->task, policy->related_cpus);
                wake_up_process(gd->task);
                init_irq_work(&gd->irq_work, cpufreq_sched_irq_work);
        }

        set_sched_freq();

        return 0;

err:
        policy->governor_data = NULL;
        kfree(gd);
        return -ENOMEM;
```

```
}
```

∗ ∗ ∗

```
static void cpufreq_sched_policy_exit(struct cpufreq_policy *policy)
{
        struct gov_data *gd = policy->governor_data;

        clear_sched_freq();
        if (cpufreq_driver_slow) {
                kthread_stop(gd->task);
                put_task_struct(gd->task);
        }

        sysfs_remove_group(&policy->kobj, get_sysfs_attr());

        policy->governor_data = NULL;

        kfree(gd);
}
```

∗ ∗ ∗

```
static int cpufreq_sched_start(struct cpufreq_policy *policy)
{
        int cpu;

        for_each_cpu(cpu, policy->cpus)
                per_cpu(enabled, cpu) = 1;

        return 0;
}
```

∗ ∗ ∗

```
static void cpufreq_sched_limits(struct cpufreq_policy *policy)
{
        unsigned int clamp_freq;
        struct gov_data *gd = policy->governor_data;;

        pr_debug("limit event for cpu %u: %u - %u kHz, currently %u kHz\n",
                policy->cpu, policy->min, policy->max,
                policy->cur);

        clamp_freq = clamp(gd->requested_freq, policy->min, policy->max);

        if (policy->cur != clamp_freq)
                __cpufreq_driver_target(policy, clamp_freq, CPUFREQ_RELATION_L);
}
```

* * *

```
static void cpufreq_sched_stop(struct cpufreq_policy *policy)
{
	int cpu;

	for_each_cpu(cpu, policy->cpus)
		per_cpu(enabled, cpu) = 0;
}
```

* * *

```
static ssize_t show_up_throttle_nsec(struct gov_data *gd, char *buf)
{
	return sprintf(buf, "%u\n", gd->up_throttle_nsec);
}
```

* * *

```
static ssize_t store_up_throttle_nsec(struct gov_data *gd,
		const char *buf, size_t count)
{
	int ret;
	long unsigned int val;

	ret = kstrtoul(buf, 0, &val);
	if (ret < 0)
		return ret;
	gd->up_throttle_nsec = val;
	return count;
}
```

* * *

```
static ssize_t show_down_throttle_nsec(struct gov_data *gd, char *buf)
{
	return sprintf(buf, "%u\n", gd->down_throttle_nsec);
}
```

* * *

```
static ssize_t store_down_throttle_nsec(struct gov_data *gd,
		const char *buf, size_t count)
{
	int ret;
	long unsigned int val;

	ret = kstrtoul(buf, 0, &val);
```

```
        if (ret < 0)
                return ret;
        gd->down_throttle_nsec = val;
        return count;
}
```