Overview of the CPU Frequency Scaling API

Introducing the CPU “Governor” Concept

Bill Gatliff
`bgat@billgatliff.com`

Freelance Embedded Systems Developer

CPU Frequency Scaling

Adjust the CPU speed:

- Increase for more processing throughput
- Decrease to lower power consumption
- Even more effective with voltage scaling

CPU Frequency Scaling

```
struct cpufreq_governor
struct cpufreq_policy
struct cpufreq_driver
```

CPU Frequency Scaling

“Governors” adjust speed:

- Per user request, or
- For maximum for performance, or
- For maximum battery life, or
- To match battery life to performance

But what defines “best”?

CPU Frequency Scaling

Adjust the CPU speed:

The question is, “to where?”

```
# ls /sys/devices/system/cpu/cpu0/cpufreq
# ls /sys/devices/system/cpu/cpu1/cpufreq
...
```

Governors

“Performance”

- Sets the processor to the highest allowable speed

“Powersave”

- Sets the processor to the lowest allowable speed

Governors

“Ondemand”

- Increase processor speed with scheduler loading

“Conservative”

- Same as “ondemand”, but ramps speed changes instead of stepping

Governors

"Userspace"

- Allow root users to specify processor speed

`/sys/.../cpufreq/scaling_setspeed`

- Userspace governor only
- Shows current CPU speed
- Write to change CPU speed
- Subject to rounding and policy limits

“What Governors Do I Have?”

`/sys/.../cpufreq/scaling_available_governors`

- Lists available governors

`/sys/.../cpufreq/scaling_governor`

- Name of currently active governor
- Write a new name to change
- (Not all governors are supported on all CPUs)

CPU Frequency Policy

```
/sys/.../cpufreq/scaling_min_freq
/sys/.../cpufreq/scaling_max_freq
```

- Maximum and minimum limits (policy)

```
/sys/.../cpufreq/cpuinfo_min_freq
/sys/.../cpufreq/cpuinfo_max_freq
```

- Maximum and minimum limits (hardware)

CPU Frequency Policy

```
/sys/.../cpufreq/scaling_driver
```

- Name of the driver implementing scaling

struct cpufreq_driver

Adjusts CPU frequency:

- The “mechanism”
- SoC-dependent
- The target frequency is decided elsewhere

struct cpufreq_driver

```
struct cpufreq_driver {
  struct module *owner;
  char name[CPUFREQ_NAME_LEN];

  u8 flags;

  int  (*init)   (struct cpufreq_policy *policy);
  int  (*verify) (struct cpufreq_policy *policy);
  ...
```

struct cpufreq_driver

```
...
/* define one out of two */
int  (*setpolicy)(struct cpufreq_policy *policy);
int  (*target)   (struct cpufreq_policy *policy,
                  unsigned int target_freq,
                  unsigned int relation);

unsigned int (*get) (unsigned int cpu);
...
```

struct cpufreq_driver

```
  ...
  int   (*exit)   (struct cpufreq_policy *policy);
  int   (*suspend)(struct cpufreq_policy *policy);
  int   (*resume) (struct cpufreq_policy *policy);
  struct freq_attr  **attr;
};
```

struct cpufreq_driver.flags

CPUFREQ_STICKY

- Driver remains even if ->init() fails (typical)

CPUFREQ_CONST_LOOPS

- loops_per_jiffy doesn't change with cpufreq

.../mach-msm/cpufreq.c

```
struct cpufreq_driver msm_cpufreq_driver = {
  .flags      = CPUFREQ_STICKY
                | CPUFREQ_CONST_LOOPS,
  .init       = msm_cpufreq_init,
  .verify     = msm_cpufreq_verify,
  .target     = msm_cpufreq_target,
  .get        = msm_cpufreq_get_freq,
  .name       = "msm",
  .attr       = msm_freq_attr,
};
```

msm_cpufreq_init()

```
int msm_cpufreq_init(...)
{
  ...
  table = cpufreq_frequency_get_table(policy->cpu);
  ...
  /*
   * In 8625 both cpu core’s frequency can not
   * be changed independently. Each cpu is bound to
   * same frequency. Hence set the cpumask to all cpu.
   */
  if (cpu_is_msm8625())
      cpumask_setall(policy->cpus);
  ...
```

msm_cpufreq_init()

```
...
if (cpufreq_frequency_table_cpuinfo(policy, table)) {
  policy->cpuinfo.min_freq = CONFIG_MSM_CPU_FREQ_MIN;
  policy->cpuinfo.max_freq = CONFIG_MSM_CPU_FREQ_MAX;
}
policy->min = CONFIG_MSM_CPU_FREQ_MIN;
policy->max = CONFIG_MSM_CPU_FREQ_MAX;
...
```

msm_cpufreq_init()

```
...
cur_freq = acpuclk_get_rate(policy->cpu);
if (cpufreq_frequency_table_target(policy,
        table, cur_freq,
        CPUFREQ_RELATION_H, &index) &&
    cpufreq_frequency_table_target(policy,
        table, cur_freq,
        CPUFREQ_RELATION_L, &index)) {
          return -EINVAL;
}
```

msm_cpufreq_init()

```
  ...
  policy->cur = cur_freq;
  policy->cpuinfo.transition_latency =
    acpuclk_get_switch_time() * NSEC_PER_USEC;

#ifdef CONFIG_SMP
    cpu_work = &per_cpu(cpufreq_work, policy->cpu);
    INIT_WORK(&cpu_work->work, set_cpu_work);
    init_completion(&cpu_work->complete);
#endif
  ...
```

msm_cpufreq_verify()

```
int msm_cpufreq_verify(struct cpufreq_policy *policy)
{
  cpufreq_verify_within_limits(policy,
     policy->cpuinfo.min_freq,
     policy->cpuinfo.max_freq);
  return 0;
}
```

msm_cpufreq_target()

```
int msm_cpufreq_target(struct cpufreq_policy *policy,
                       unsigned int target_freq,
                       unsigned int relation)
{
  ...
  table = cpufreq_frequency_get_table(policy->cpu);
  if (cpufreq_frequency_table_target(policy,
          table, target_freq, relation, &index)) {
            ret = -EINVAL;
            goto done;
  }
  ...
```

msm_cpufreq_target()

```
#ifdef CONFIG_SMP
  cpu_work = &per_cpu(cpufreq_work, policy->cpu);
  cpu_work->policy = policy;
  cpu_work->frequency = table[index].frequency;
  cpu_work->status = -ENODEV;

  ...
  ret = set_cpu_freq(cpu_work->policy,
                     cpu_work->frequency);
  ...
```

struct cpufreq_policy

Captures CPU frequency policy:

- What the target frequency is
- What the limits are
- The target frequency is decided elsewhere

struct cpufreq_policy

```
struct cpufreq_policy {
  /* CPUs sharing clock, require sw coordination */
  cpumask_var_t cpus;   /* Online CPUs only */
  cpumask_var_t related_cpus; /* Online + Offline CPUs */
  ...
  unsigned int  cpu; /* CPU managing this policy */
  ...
```

struct cpufreq_policy

```
  ...
  unsigned int  min;    /* in kHz */
  unsigned int  max;    /* in kHz */
  unsigned int  cur;    /* in kHz, for governors */
  ...
  struct cpufreq_governor *governor;
};

DEFINE_PER_CPU(struct cpufreq_policy *, cpufreq_cpu_data);
```

struct cpufreq_governor

Defines the target CPU frequency:

- ... within policy limits
- ... according to a governing algorithm

Doesn’t know how the target is achieved

struct cpufreq_governor

```
struct cpufreq_governor {
  char name[CPUFREQ_NAME_LEN];
  ...
  int (*governor)(struct cpufreq_policy *policy,
                  unsigned int event);
  ...
  struct list_head        governor_list;
  struct module           *owner;
};
```

struct cpufreq_governor

```
struct cpufreq_governor my_gov = {
  .name     = "my-cpufreq-gov",
  .governor = my_cpufreq_gov,
  .owner    = THIS_MODULE,
};
...
cpufreq_register_governor(&my_gov);
...
```

cpufreq_gov_performance

```
int cpufreq_gov_performance(...)
{
  switch (event) {
    case CPUFREQ_GOV_START:
    case CPUFREQ_GOV_LIMITS:
      __cpufreq_driver_target(policy, policy->max,
                              CPUFREQ_RELATION_H);
      break;
    default:
      break;
  }
  return 0;
}
```

cpufreq_gov_performance

```
int __cpufreq_driver_target(...)
{
  ...
  if (target_freq > policy->max)
     target_freq = policy->max;
  ...
  if (target_freq == policy->cur)
     return 0;
  ...
  ret = cpufreq_driver->target(policy,
                     target_freq, relation);
  ...
}
```

cpufreq_gov_performance

```
int cpufreq_driver_target(...)
{
  policy = cpufreq_cpu_get(policy->cpu);
  ...
  ret = __cpufreq_driver_target(policy,
                target_freq, relation);
  ...
  cpufreq_cpu_put(policy);
  ...
}
```

cpufreq_gov_ondemand

```
void dbs_check_cpu(...)
{
  /* Extrapolated load of this CPU */
  unsigned int load_at_max_freq = 0;
  unsigned int avg_load_at_max_freq = 0;
  unsigned int max_load_freq;
  /* Current load across this CPU */
  unsigned int cur_load = 0;
  ...
```

cpufreq_gov_ondemand

```
...
idle_time = get_cpu_idle_time(...);
iowait_time = get_cpu_iowait_time(...);
...
```