Power allocator governor tunables

Trip points

The governor works optimally with the following two passive trip points:

1. "switch on" trip point: temperature above which the governor control loop starts operating. This is the first passive trip point of the thermal zone.
2. "desired temperature" trip point: it should be higher than the "switch on" trip point. This the target temperature the governor is controlling for. This is the last passive trip point of the thermal zone.

PID Controller

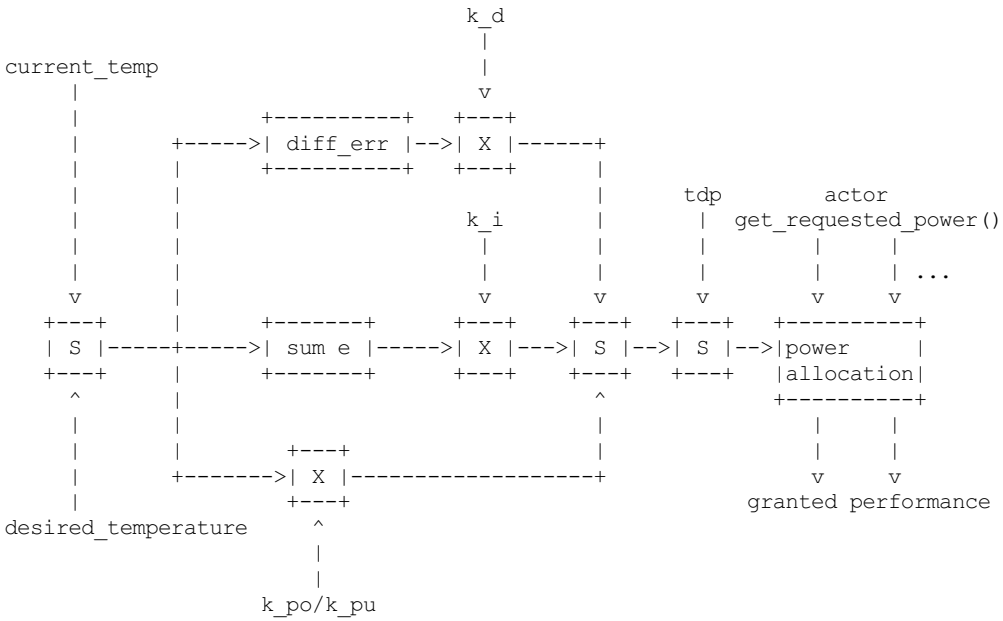
The power allocator governor implements a Proportional-Integral-Derivative controller (PID controller) with temperature as the control input and power as the controlled output:

P_max = k_p * e + k_i * err_integral + k_d * diff_err + sustainable_power

where

- e = desired_temperature - current_temperature
- err_integral is the sum of previous errors
- diff_err = e - previous_error

It is similar to the one depicted below:



Sustainable power

An estimate of the sustainable dissipatable power (in mW) should be provided while registering the thermal zone. This estimates the sustained power that can be dissipated at the desired control temperature. This is the maximum sustained power for allocation at the desired maximum temperature. The actual sustained power can vary for a number of reasons. The closed loop controller will take care of variations such as environmental conditions, and some factors related to the speed-grade of the silicon. *sustainable_power* is therefore simply an estimate, and may be tuned to affect the aggressiveness of the thermal ramp. For reference, the sustainable power of a 4" phone is typically 2000mW, while on a 10" tablet is around 4500mW (may vary depending on screen size). It is possible to have the power value expressed in an abstract scale. The sustained power should be aligned to the scale used by the related cooling devices.

If you are using device tree, do add it as a property of the thermal-zone. For example:

```
thermal-zones {
        soc_thermal {
                polling-delay = <1000>;
                polling-delay-passive = <100>;
                sustainable-power = <2500>;
                ...
```

Instead, if the thermal zone is registered from the platform code, pass a *thermal_zone_params* that has a *sustainable_power*. If no *thermal_zone_params* were being passed, then something like below will suffice:

```
static const struct thermal_zone_params tz_params = {
        .sustainable_power = 3500,
};
```

and then pass *tz_params* as the 5th parameter to *thermal_zone_device_register()*

k_po and k_pu

The implementation of the PID controller in the power allocator thermal governor allows the configuration of two proportional term constants: *k_po* and *k_pu*. *k_po* is the proportional term constant during temperature overshoot periods (current temperature is above "desired temperature" trip point). Conversely, *k_pu* is the proportional term constant during temperature undershoot periods (current temperature below "desired temperature" trip point).

These controls are intended as the primary mechanism for configuring the permitted thermal "ramp" of the system. For instance, a lower *k_pu* value will provide a slower ramp, at the cost of capping available capacity at a low temperature. On the other hand, a high value of *k_pu* will result in the governor granting very high power while temperature is low, and may lead to temperature overshooting.

The default value for *k_pu* is:

```
2 * sustainable_power / (desired_temperature - switch_on_temp)
```

This means that at *switch_on_temp* the output of the controller's proportional term will be 2 * *sustainable_power*. The default value for *k_po* is:

```
sustainable_power / (desired_temperature - switch_on_temp)
```

Focusing on the proportional and feed forward values of the PID controller equation we have:

```
P_max = k_p * e + sustainable_power
```

The proportional term is proportional to the difference between the desired temperature and the current one. When the current temperature is the desired one, then the proportional component is zero and *P_max = sustainable_power*. That is, the system should operate in thermal equilibrium under constant load. *sustainable_power* is only an estimate, which is the reason for closed-loop control such as this.

Expanding *k_pu* we get:

```
P_max = 2 * sustainable_power * (T_set - T) / (T_set - T_on) +
    sustainable_power
```

where:

- T_set is the desired temperature
- T is the current temperature
- T_on is the switch on temperature

When the current temperature is the switch_on temperature, the above formula becomes:

```
P_max = 2 * sustainable_power * (T_set - T_on) / (T_set - T_on) +
    sustainable_power = 2 * sustainable_power + sustainable_power =
    3 * sustainable_power
```

Therefore, the proportional term alone linearly decreases power from 3 * *sustainable_power* to *sustainable_power* as the temperature rises from the switch on temperature to the desired temperature.

k_i and integral_cutoff

k_i configures the PID loop's integral term constant. This term allows the PID controller to compensate for long term drift and for the quantized nature of the output control: cooling devices can't set the exact power that the governor requests. When the temperature error is below *integral_cutoff*, errors are accumulated in the integral term. This term is then multiplied by *k_i* and the result added to the output of the controller. Typically *k_i* is set low (1 or 2) and *integral_cutoff* is 0.

k_d

k_d configures the PID loop's derivative term constant. It's recommended to leave it as the default: 0.

Cooling device power API

Cooling devices controlled by this governor must supply the additional "power" API in their *cooling_device_ops*. It consists on three ops:

1.
```
int get_requested_power(struct thermal_cooling_device *cdev,
                        struct thermal_zone_device *tz, u32 *power);
```

@cdev:
The *struct thermal_cooling_device* pointer
@tz:
thermal zone in which we are currently operating
@power:
pointer in which to store the calculated power

get_requested_power() calculates the power requested by the device in milliwatts and stores it in @power . It should return 0 on success, -E* on failure. This is currently used by the power allocator governor to calculate how much power to give to each cooling device.

2.
```
int state2power(struct thermal_cooling_device *cdev, struct
                thermal_zone_device *tz, unsigned long state,
                u32 *power);
```

@cdev:
The *struct thermal_cooling_device* pointer
@tz:
thermal zone in which we are currently operating
@state:
A cooling device state
@power:
pointer in which to store the equivalent power

Convert cooling device state @state into power consumption in milliwatts and store it in @power. It should return 0 on success, -E* on failure. This is currently used by thermal core to calculate the maximum power that an actor can consume.

3.
```
int power2state(struct thermal_cooling_device *cdev, u32 power,
                unsigned long *state);
```

@cdev:
The *struct thermal_cooling_device* pointer
@power:
power in milliwatts
@state:
pointer in which to store the resulting state

Calculate a cooling device state that would make the device consume at most @power mW and store it in @state. It should return 0 on success, -E* on failure. This is currently used by the thermal core to convert a given power set by the power allocator governor to a state that the cooling device can set. It is a function because this conversion may depend on external factors that may change so this function should the best conversion given "current circumstances".

Cooling device weights

Weights are a mechanism to bias the allocation among cooling devices. They express the relative power efficiency of different cooling devices. Higher weight can be used to express higher power efficiency. Weighting is relative such that if each cooling device has a weight of one they are considered equal. This is particularly useful in heterogeneous systems where two cooling devices may perform the same kind of compute, but with different efficiency. For example, a system with two different types of processors.

If the thermal zone is registered using *thermal_zone_device_register()* (i.e., platform code), then weights are passed as part of the thermal zone's *thermal_bind_parameters*. If the platform is registered using device tree, then they are passed as the *contribution* property of each map in the *cooling-maps* node.

Limitations of the power allocator governor

The power allocator governor's PID controller works best if there is a periodic tick. If you have a driver that calls *thermal_zone_device_update()* (or anything that ends up calling the governor's *throttle()* function) repetitively, the governor response won't be very good. Note that this is not particular to this governor, step-wise will also misbehave if you call its throttle() faster than the normal thermal framework tick (due to interrupts for example) as it will overreact.

Energy Model requirements

Another important thing is the consistent scale of the power values provided by the cooling devices. All of the cooling devices in a single thermal zone should have power values reported either in milli-Watts or scaled to the same 'abstract scale'.