Intelligent Power Allocator – Tuning Guide

What is IPA?

Concept

- Temperature is proportional to power consumption
- Power is consumed by the various components in the SoC
- Controlling device power consumption can be used to control temperature
- Intelligently allocating power should maximise performance

What is IPA?

Components - Power allocator thermal governor

- Proportional Integral Derivative (PID) Controller
- Inputs
  - Configuration Parameters
  - Temperature
  - Requested Power
- Outputs power grants
- Integrates with the Linux Thermal sub-system

What is IPA?

Components - Cooling devices

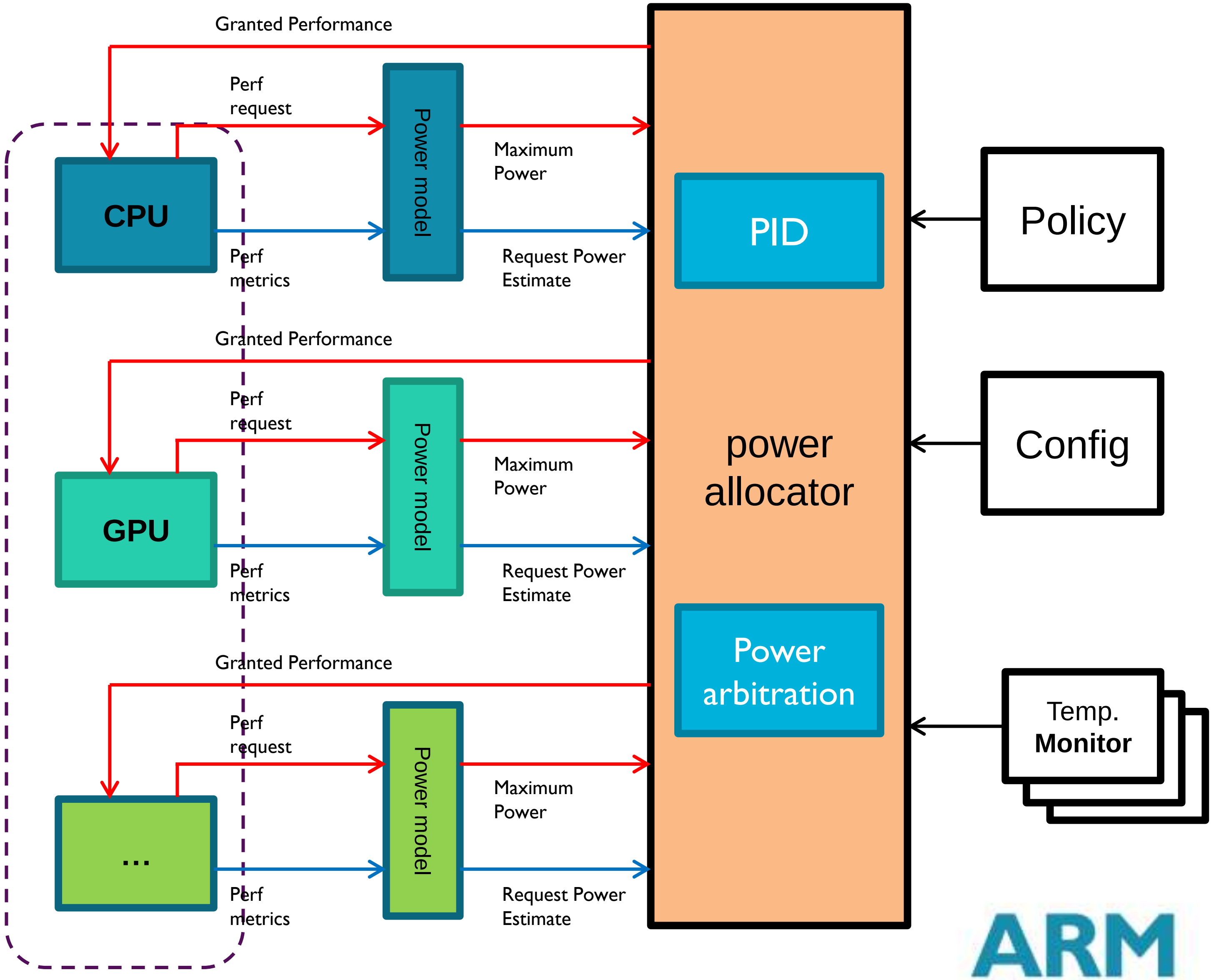
- Implement device power model
  - ‘big’, ‘LITTLE’, ‘GPU’, etc., are devices
- Can restrict device power consumption
- Translate power <-> performance cap
- Meaning of performance is device dependent
  - Frequency for cpufreq and devfreq cooling devices
  - Could be charging current for battery or brightness for LCD
- Need to support power extensions API to work with power allocator governor
  - requested_power, state2power, power2state
  - Without these, it’s a legacy cooling device that works with other governors
- Implementation already available for
  - cpu cooling
  - devfreq cooling

Algorithm - Overview

Governor performs two tasks:

1. Keeps **system within thermal envelope**
   - Proactive power budget via **closed loop control**
   - **Exploits thermal headroom**

2. **Dynamic power allocation per device**
   - Performance demand & power models
   - Power divided based on what each device requested. Anything left over is distributed among the devices, up to their maximum.



ARM

Algorithm

- PID controller estimates available thermal headroom (power budget) using previous and current temperatures
  - Assumes the controller is driven at uniform time intervals
- Error e = control temperature – current temperature
- estimated_power = k_p * e + k_i * ∑e + k_d * d/dt (e)
- output = sustainable_power + estimated_power
- Output is allocated to requesting devices based on their power requests
- Cooling device translate power grant to performance caps

PID Parameters

- Proportional
  - k_pu – when temperature is under control temperature, $e > 0$
  - k_po – when temperature is over control temperature, $e < 0$
- Integral k_i
  - Only used when $e <$ integral_cutoff
  - Integral_cutoff is 0 by default
- Derivative k_d
  - Currently unused

Implementing IPA on your platform

Setup

Prerequisites

- Ensure kernel has support for power allocator thermal governor
- Available in
  - Linux v4.2+
  - Linaro LSK 3.10
  - Linaro LSK 3.18
  - Chrome OS 3.14 (in-progress)
  - Chrome OS 3.18
- Operating Points (OPPs) are used for DVFS
  - cpufreq
  - devfreq

Linaro Stable Kernels (LSK) for IPA evaluation

- https://git.linaro.org/landing-teams/working/arm/kernel-release.git
- lsk-3.18-armlt (use 'lsk-3.18-armlt-20151001' or later tag)
- lsk-3.10-armlt-juno (choose 'lsk-3.10-armlt-juno-20150424' or later tag)

- Contains IPA + GTS integrated with Juno port

- Includes example power cooling device implementations
  - Includes devfreq based integration of GPU (Mali Midgard)
  - Includes Juno porting

Setup

Cooling devices

- cpu cooling for CPUs
- devfreq cooling for GPUs
- Provide dynamic power coefficient
  - API call
  - device tree?
  - Used to calculate dynamic power coefficient
  - P_{dyn} = dynamic-power-coefficient * V^2 * f
- If significant, static power consumption can be provided via callback
  - Recommend to initially concentrate on dynamic power model

CPU Power Cooling Device

Static Power

- Static leakage power consumption depends on a number of factors
  => needs subroutine to calculate
  - Time the circuit spends in each 'power state‘ (e.g. ON, OFF, ‘retention’)
  - Power managed regions
  - Temperature
  - Voltage
  - Process (e.g. slow to fast – possibly varies on specific chips – bins etc)
  - Type, number and size of transistors in both the logic gates and any RAM elements included.
    (e.g. LVt vs. HVt transistors)

- Note: OS awareness of factors affecting static power can vary, so the static model is likely to be platform specific

Example - Juno platform static power function

- Juno model is simple
  - Number of CPUs ON
  - Temperature
  - Voltage
  - Cache leakage if cluster ON

IPA doesn't currently track the residency of the CPUs in the different idle states and account for the static power accordingly

```
/* voltage in uV */
static int get_static_power(cpumask_t *cpumask, int interval,
                            unsigned long u_volt, u32 *static_power)
{
        unsigned long temperature, t_scale, v_scale;
        u32 cpu_coeff;
        int nr_cpus = cpumask_weight(cpumask);
        enum cluster_type cluster =
                topology_physical_package_id(cpumask_any(cpumask));

        if (!scpi_temp_sensor.tzd)
                return -ENODEV;

        cpu_coeff = cluster_data[cluster].static_coeff;

        /* temperature in mC */
        temperature = scpi_temp_sensor.tzd->temperature / 1000;

        t_scale = get_temperature_scale(temperature);
        v_scale = get_voltage_scale(u_volt);

        *static_power = nr_cpus * (cpu_coeff * t_scale * v_scale) / 1000000;

        if (nr_cpus) {
                u32 cache_coeff = cluster_data[cluster].cache_static_coeff;

                /* cache leakage */
                *static_power += (cache_coeff * v_scale * t_scale) / 1000000;
        }

        return 0;
}
```

CPU Power Cooling Device

Static Power - registering

- Pass a function pointer that follows the 'get_static_t' prototype:

-
  ```
  int plat_get_static(cpumask_t *cpumask, int interval, unsigned long voltage,
                      u32 *power);
  ```

  - `cpumask` a mask of the cpus involved in the calculation,
  - `interval` is the last period (usually the power allocator period)
  - `voltage` the voltage at which they are operating (in microvolts – uV)
  - `power` output variable, the function should write the static power there

- If 'plat_static_func' is NULL when registering the power cpu cooling device, static power is considered to be negligible for this platform and only dynamic power is considered.

CPU Power Cooling Device

Static Power - calculation

- The platform specific callback can then use any combination of tables and/or equations to calculate the static power. [this can include platform or device/bin-specific process grade data if available]

- Note: the significance of static power for CPUs in comparison to dynamic power is highly dependent on implementation. Given the potential complexity in implementation, the importance and accuracy of its inclusion when using cpu cooling devices should be assessed on a case by cases basis.
  - Juno we use static power calculation
  - Exynos (HKMG process) we don’t use static power

Mali GPU Power Cooling Device

- The Mali power cooling device is supported in the r5p0 version of the DDK for Midgard GPUs
- Uses devfreq
- Assumes GPU is a large block of logic that all runs at specified voltage/frequency
- drivers/gpu/arm/midgard/mali_kbase_devfreq.c
- Being ported to Utgard as well

Setup

Thermal zone

- Setup thermal zone with appropriate sensor
  - Representative sensor (such as SoC sensor or skin sensor) or a combination of device sensors
- Provide sustainable power
  - Device tree
  - Thermal zone parameters
- Enable CONFIG_THERMAL_GOV_POWER_ALLOCATOR
- Set policy to power allocator. Can be done via
  - Compile time default, CONFIG_THERMAL_DEFAULT_GOV_POWER_ALLOCATOR
  - Setup in userspace at boot time by writing 'power_alloator' to policy node in thermal zone
- Setup two passive trip points
  - first trip point is "switch on" temperature
  - Second trip point is "control" temperature
- Bind cooling devices to "control" trip point

Setup

CONFIG ENTRIES

- Upstream version of IPA is implemented as a new governor, called 'power allocator.'

CONFIG_THERMAL_GOV_POWER_ALLOCATOR	Enables power allocator governor
CONFIG_THERMAL_DEFAULT_GOV_POWER_ALLOCATOR	Makes power allocator the default governor
CONFIG_DEVFREQ_THERMAL	Enables thermal management for devfreq devices (used by Mali Thermal driver)
CONFIG_MALI_DEVFREQ	Enabled devfreq for Mali
CONFIG_CPU_THERMAL	Cpu cooling device
CONFIG_DEVFREQ_THERMAL	Devfreq cooling device
CONFIG_SCPI_THERMAL	*Enables the Juno Platform thermal integration*
CONFIG_PM_OPP	Enable Operating Performance Point (OPP) library
CONFIG_DEBUG_FS	Recommended during tuning for /d files
CONFIG_THERMAL_WRITABLE_TRIPS	Change trip points from sysfs, useful for tuning

Power allocator specific configs

Juno specific

Other useful generic configs

ARM

No DT registering thermal_zone and trip points

- Registering thermal_zone_device with estimate of the max sustainable power (in mW). Use ‘thermal_zone_params’ that have a ‘sustainable_power’. e.g. pass ‘tz_params’ as the 5th parameter to ‘thermal_zone_device_register()’

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

- Other parameters (k_po, k_pu,…) can be passed as thermal_zone_params
- Trip points can be either active or passive
  - **"switch on"** - temperature above which the governor control loop starts operating
  - **"desired temperature"** - PID target temperature
- Bind() op of the thermal zone can now include the weight of the cooling device

DT registering thermal_zone and trip points

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

                thermal-sensors = <&scpi_sensor0 3>;

                trips {
                        threshold: trip-point@0 {
                                temperature = <55000>;
                                hysteresis = <1000>;
                                type = "passive";
                        };
                        target: trip-point@1 {
                                temperature = <65000>;
                                hysteresis = <1000>;
                                type = "passive";
                        };
                };

                cooling-maps {
                        map0 {
                            trip = <&target>;
                            cooling-device = <&cluster0 0 4>;
                            contribution = <1024>;
                        };
                        map1 {
                            trip = <&target>;
                            cooling-device = <&cluster1 0 4>;
                            contribution = <2048>;
                        };
                        map2 {
                           trip = <&target>;
                           cooling-device = <&gpu 0 4>;
                           contribution = <1024>;
                        };
                };
        };
};
```

Power allocator

2 passive trip points
trip-point@0
trip-point@1

Switch_on
55degC

Target temp
65degC

Currently only trip-points, sustainable-power and weights can be specified in DT

(ARM looking at adding support for the other parameters when no chance of further change)

When using DT, boot happens with defaults and userspace can change it by writing to sysfs files.

Setup

Porting parameters

sysfs location: /sys/class/thermal/thermal_zoneX/:

struct element	DT name	sysfs (writeable)	
switch_on temperature	trip-point@0	trip_point_0_temp	Temperature above which IPA starts operating (first passive trip point – trip 0)
desired_temperature	trip-point@1	trip_point_1_temp	Target temperature (last passive trip point – trip 1)
weight	contribution	cdevX_weight	Weight for the cooling device
sustainable_power	sustainable-power	sustainable_power	Max sustainable power
k_po	*[future]*	k_po	Proportional term constant during temperature overshoot periods
k_pu	*[future]*	k_pu	Proportional term constant during temperature undershoot periods
k_i	*[future]*	k_i	PID loop's integral term constant (compensates for long-term drift) When the temperature error is below 'integral_cutoff', errors are accumulated in the integral term
k_d	*[future]*	k_d	PID loop's derivative term constant (typically 0)
integral_cutoff	*[future]*	integral_cutoff	Typically 0 so cutoff not used

Setup

Validation

- Cooling devices
  - # of cooling_device* nodes in /sys/class/thermal match configuration
- Thermal zones
  - # of thermal_zone* nodes in /sys/class/thermal match configuration
  - "cat temp" in thermal_zone to check if sensor reports reasonable temperature (in mC)
  - Value of sustainable_power in thermal_zone matches configuration
  - Check trip point temperatures in trip_point_*_temp nodes
  - links to bound cooling device
  - Weights associated with cooling devices
  - Ensure cooling devices are bound to "control" trip point

Tuning

Workloads

- Identify workloads
  - Should include single as well as multiple device loads

Weights

- Bias power allocation among cooling devices in a thermal zone
  - CPU vs GPU
- Can express relative efficiency among similar cooling devices
  - E.g., big.LITTLE
  - TODO formula that equalises 'big' to 'LITTLE', or goes even further to bias towards 'LITTLE'
- For mobile, we suggest setting GPU weight equal to 'LITTLE'
  - Bias power allocation to the GPU
  - Found to give good performance for mobile workloads
  - Only a suggestion, different usecases may benefit from alternate biasing

Weights – used to prioritize cooling

- A mechanism to bias the allocation amongst cooling devices
- When cooling devices request power, that request is multiplied by the weight
  - Power-efficient devices should have higher weights
- Recommendation for typical big.LITTLE SoC:
  - Favors little cpu when limiting CPU's

	Weight / 'contribution'
big cpu	1024
little cpu	2048
GPU	1024

- Will change when EAS/IPA integrated

Methodology

- Run workload
  - Initial conditions are consistent across runs
  - Capture ‘thermal*’ trace events
  - workload automation makes this easy
- Generate report using tuning flow template

Tooling

- trace-cmd
  - git clone git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git –b trace-cmd-v2.6
  - cd trace-cmd
  - make && sudo make install
- Workload automation
  - sudo pip install wlauto[all]
- TRAPpy
  - sudo pip install trappy[notebook]
- Tuning flow

Kernel Trace

- Captures kernel trace in an in-memory buffer
- Dumped to file at the end of the run
- Sample trace -

```
kworker/7:2-4443  [007]  2107.527795: thermal_temperature:  thermal_zone=exynos-therm id=0 temp_prev=73242
temp=72997
kworker/7:2-4443  [007]  2107.527895: thermal_power_cpu_get_power: cpus=000000f0 freq=800000 load={0 0 0 0}
dynamic_power=0 static_power=0
kworker/7:2-4443  [007]  2107.527916: thermal_power_cpu_get_power: cpus=0000000f freq=1500000 load={16 52 5 24}
dynamic_power=249 static_power=0
kworker/7:2-4443  [007]  2107.527920: thermal_power_allocator_pid: thermal_zone_id=0 err=3892 err_integral=0 p=5321 i=0
d=0 output=7821
kworker/7:2-4443  [007]  2107.527960: thermal_power_cpu_limit: cpus=000000f0 freq=1900000 cdev_state=0 power=4165
kworker/7:2-4443  [007]  2107.527970: thermal_power_cpu_limit: cpus=0000000f freq=1600000 cdev_state=0 power=1100
kworker/7:2-4443  [007]  2107.527977: thermal_power_allocator: thermal_zone_id=0 req_power={2555 0 996}
total_req_power=3551 granted_power={2555 4165 1100} total_granted_power=7820 power_range=7821
max_allocatable_power=10283 current_temperature=73108 delta_temperature=3892
```

Tracing in WA

- To configure tracing in WA add the following to your agenda:
  - `instrumentation: [trace-cmd]`
  - `trace_events: [‘thermal*’]`

- If you want to run trace-cmd manually do:
  - `trace-cmd –e “thermal*”`
  - The thermal framework ones are all prefixed with “thermal_” and the power allocator ones are prefixed with “thermal_power_allocator_” so “thermal*” captures all of them.

- Also visible in debugfs: /sys/kernel/debug/tracing/events

Running workloads

- An example of a Workload Automation agenda for Juno can be seen on the right hand side
- The notebook template will be distributed with the IPA Tuning Flow (not yet done)
- For more documentation on WA see https://pythonhosted.org/wlauto/
- Run the agenda with

```
wa run /path/to/agenda.yaml
```

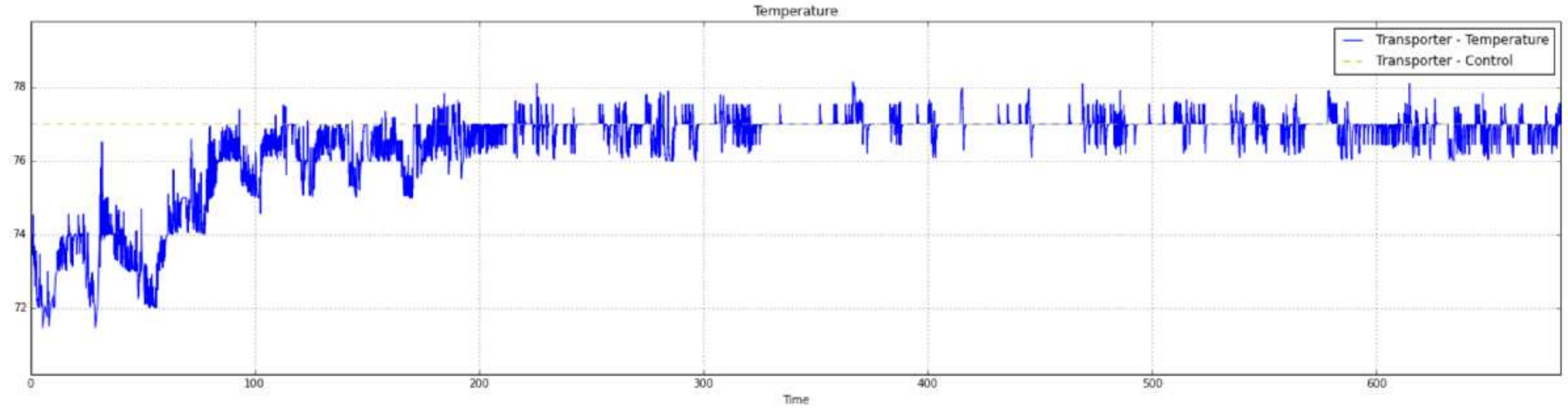
```
config:
        device: juno
        device_config:
                adb_name: 10.2.131.84:5555
        instrumentation: [fps, trace-cmd]
        result_processors: [ipynb_exporter]
        trace_events: ['thermal*']
        ipynb_exporter:
                notebook_template: /path/to/template.ipynb
                convert_to_pdf: True
                show_pdf: True

global:
        iterations: 1

workloads:
        - name: antutu
          params:
                times: 5
```

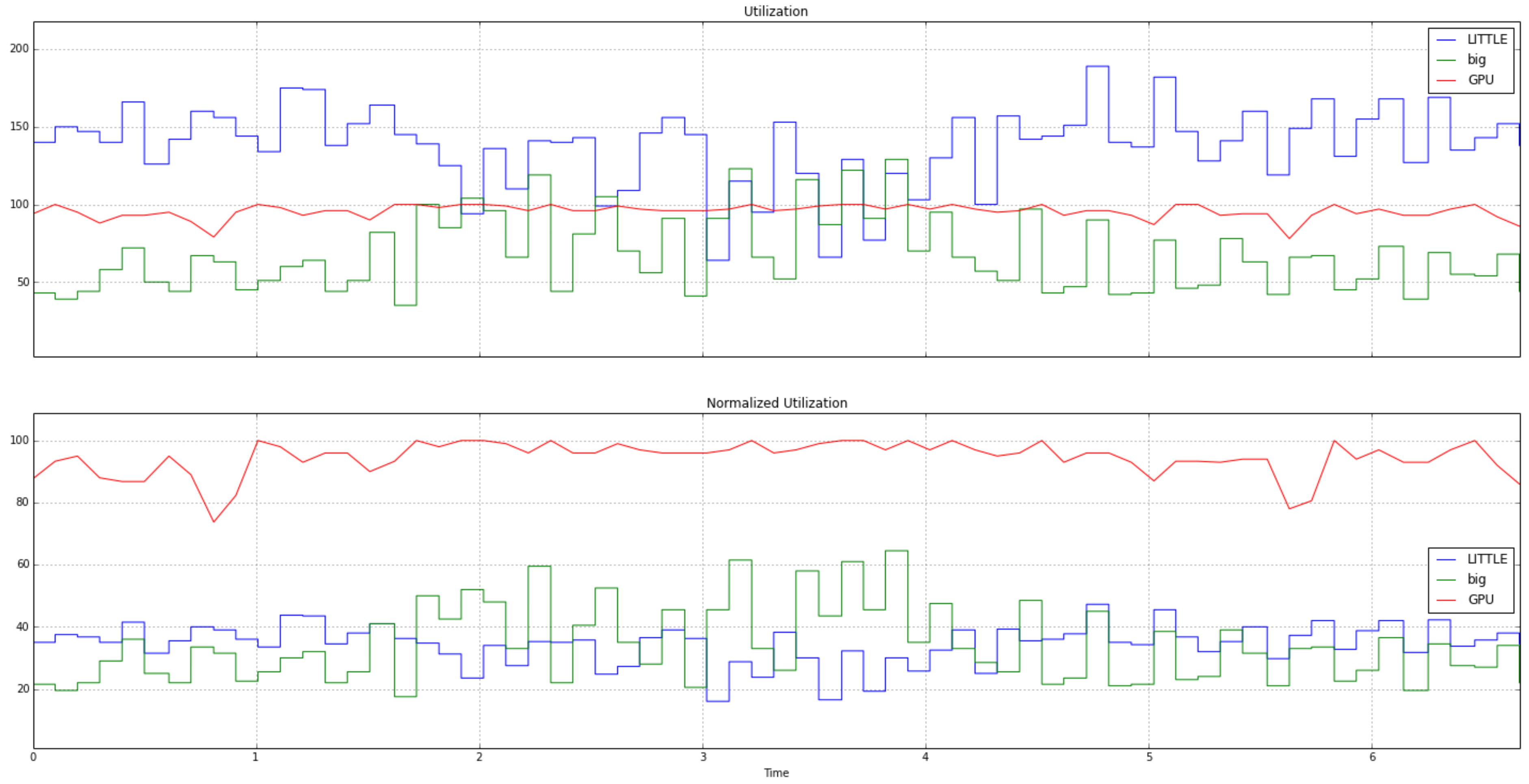
IPA Tuning flow (1)

The first graph is the temperature graph. The yellow dashed line is the control temperature, the temperature we're aiming for. The blue line is the current temperature.



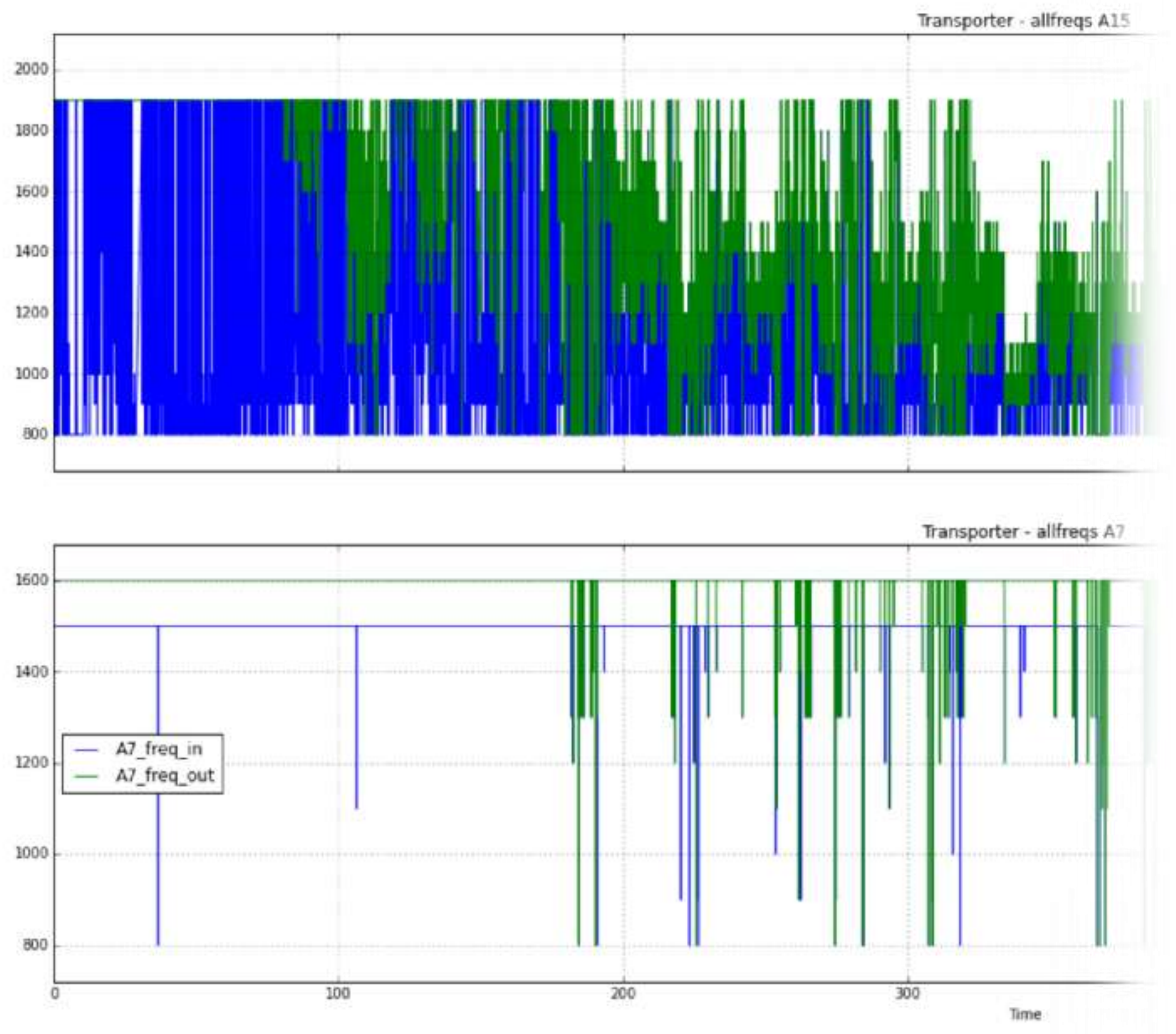
IPA Tuning flow (2)

- The Utilization plot shows the load of each of the power allocator cooling devices: big, LITTLE and GPU. Normalized utilization is $utilization * freq/maxfreq$.



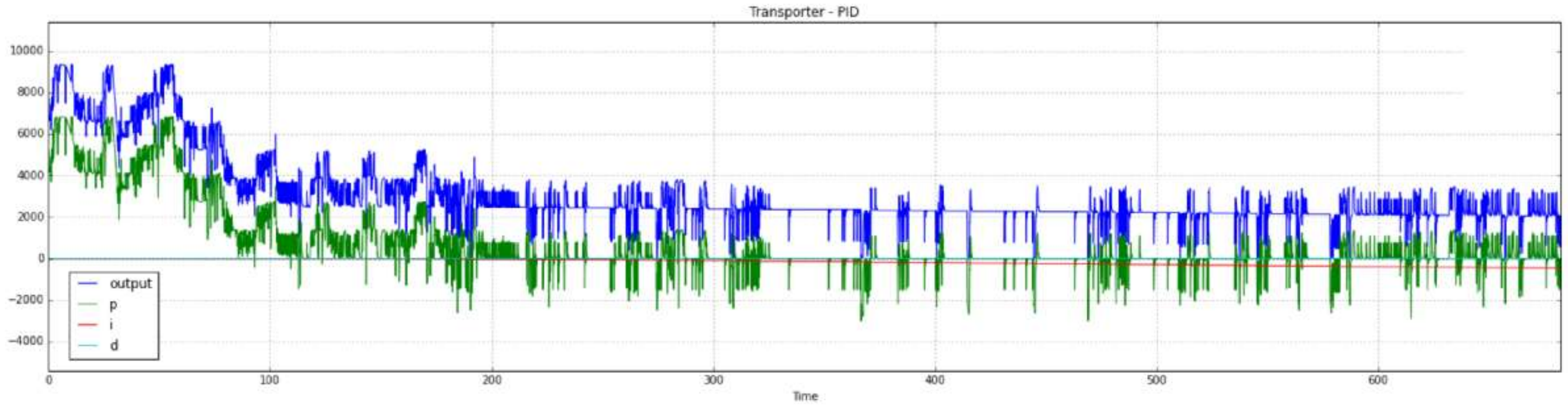
IPA Tuning flow (3)

- Allfreqs plots show the current frequency (in) and the frequency granted by the power allocator governor (out)



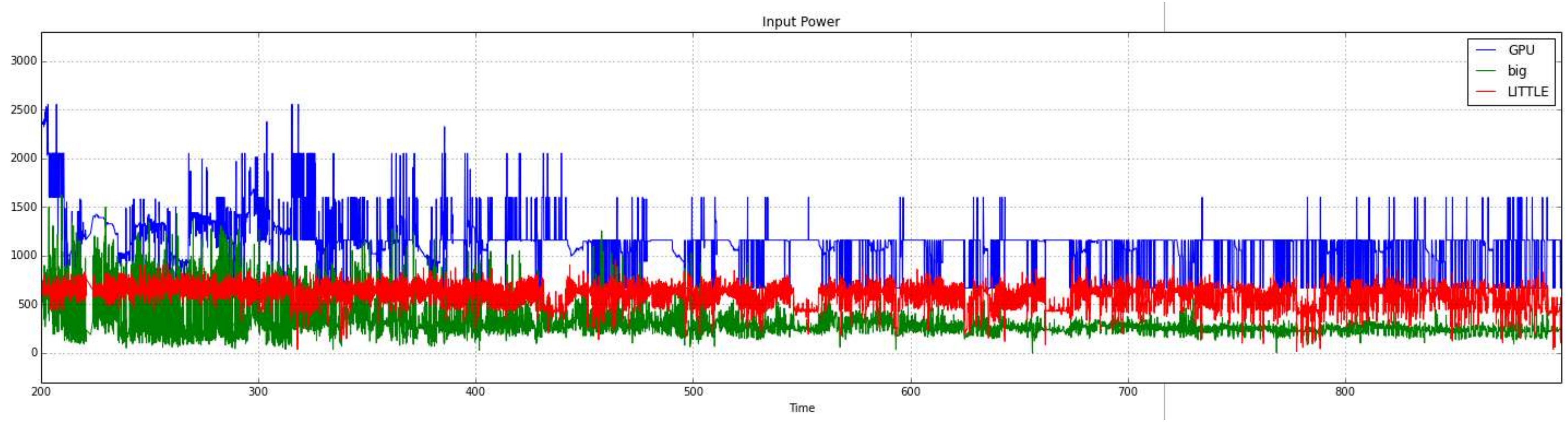
IPA Tuning flow (4)

- The PID plot shows each component of the PID algorithm and its contribution to the output (the power granted by IPA).
- It's recommended to have the output (blue line) dominated by the "p" component, with "i" component doing minor corrections.



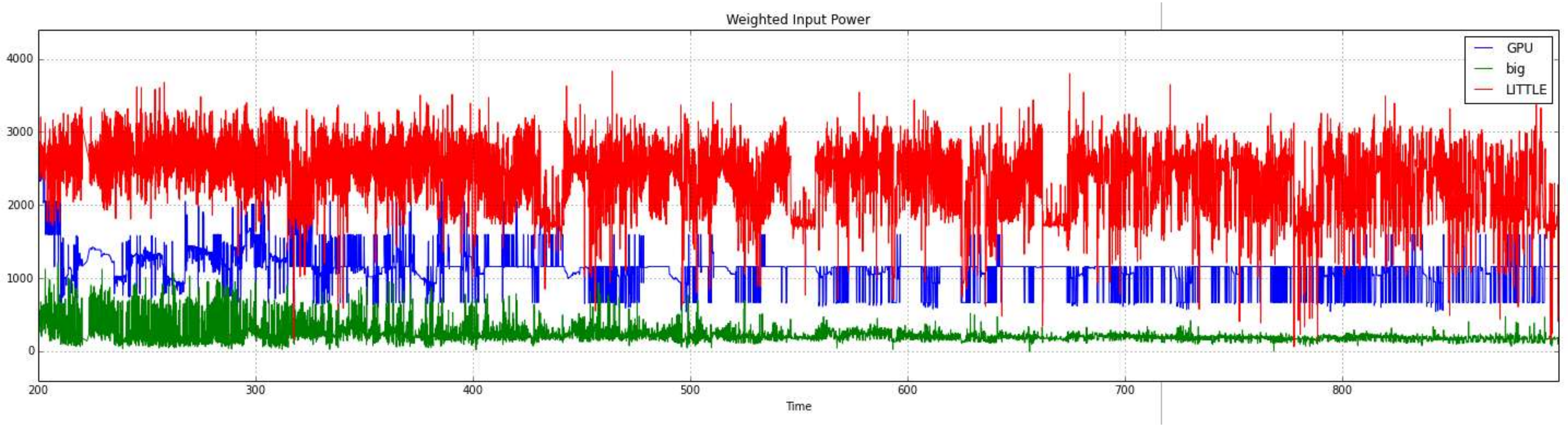
IPA Tuning flow (5)

- The input power plot shows how much power each device consumes as seen by the power allocator governor. Weight is not taken into account (see next slide).



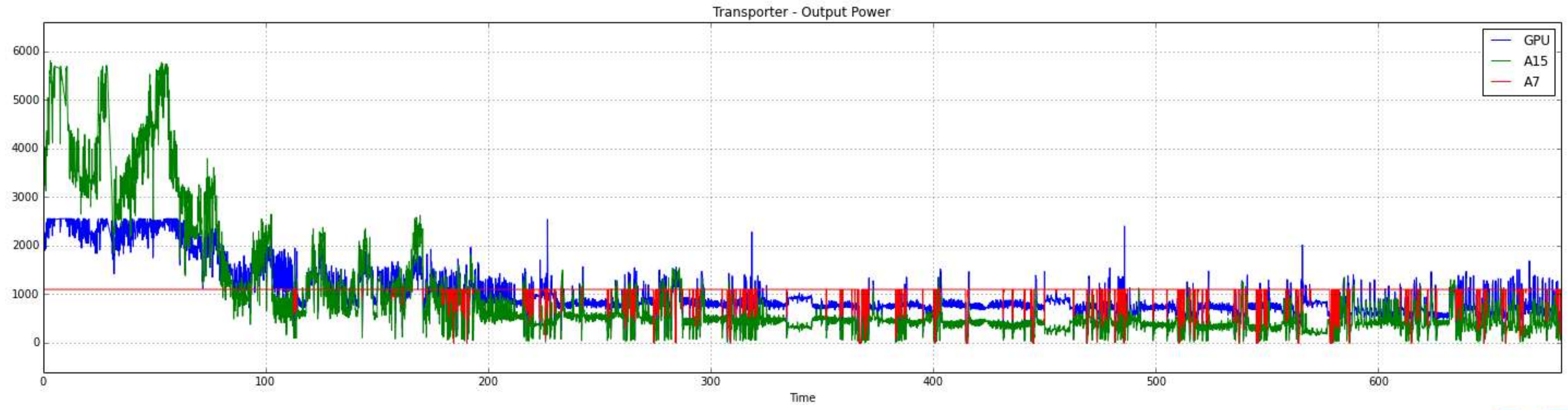
IPA Tuning flow (6)

- The weighted input power is what the governor uses to make its allocating decisions. For the plot below, the weights for big/LITTLE/GPU were 768/4096/1024, that's why the LITTLE appears to ask for so much power.



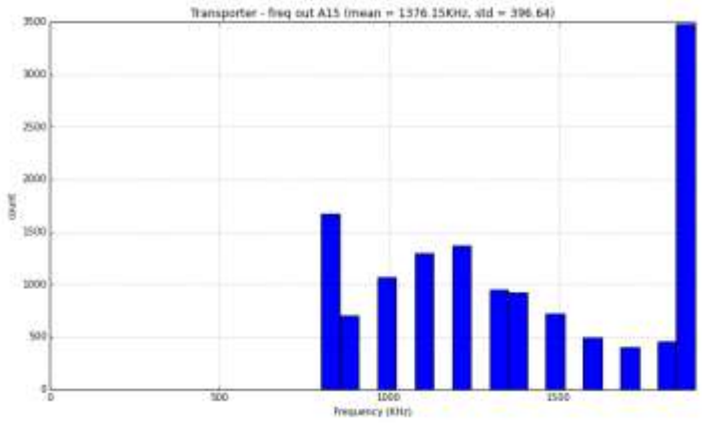
IPA Tuning flow (7)

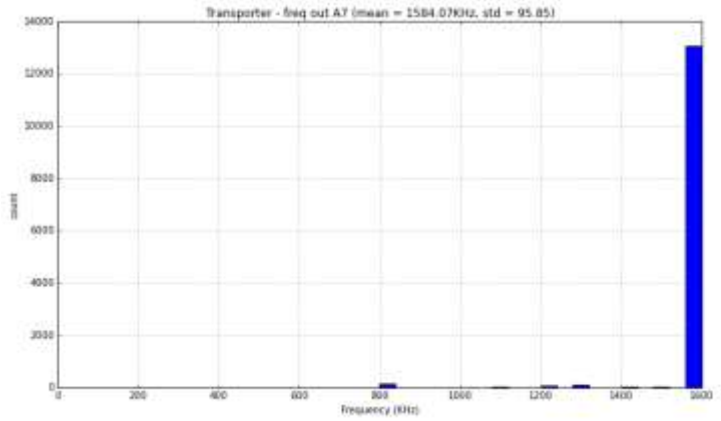
- The output of the PID ("output" in the PID plot) is divided amongst the cooling devices.
- The output power plot shows how much power the governor granted to each device.
- In the example below we see that in the beginning, when there's thermal headroom, A15 and GPU get plenty of power. As the temperature approaches the control temperature, they are constrained. The A7 gets its maximum power throughout the run as it's the most power efficient CPU. The GPU gets more power than the A15, as it's a GPU intensive workload.

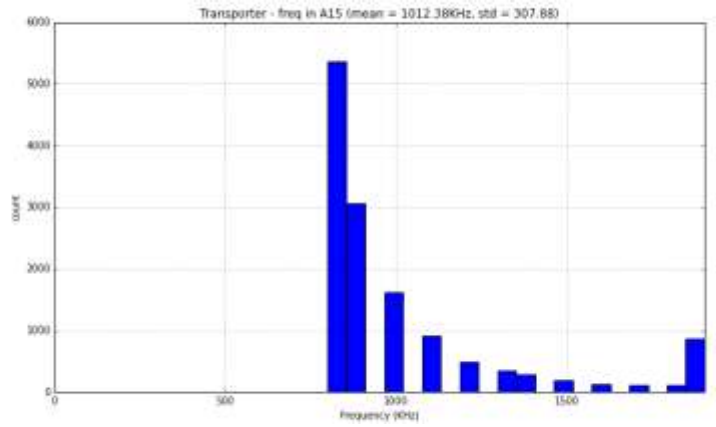


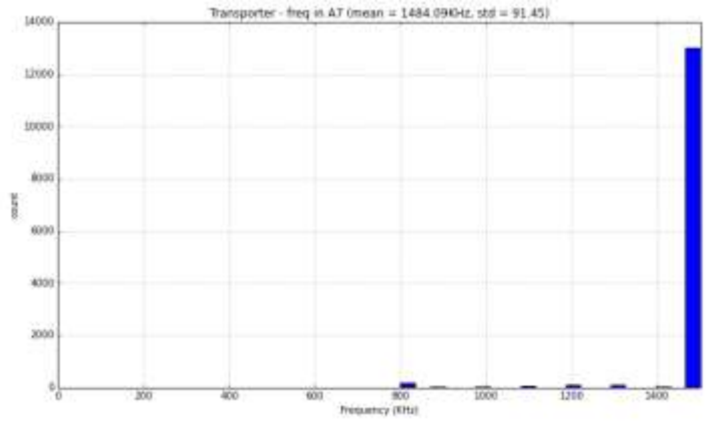
IPA Tuning flow (8)

- The frequency histograms show the same data as the allfreqs plots but as histograms.



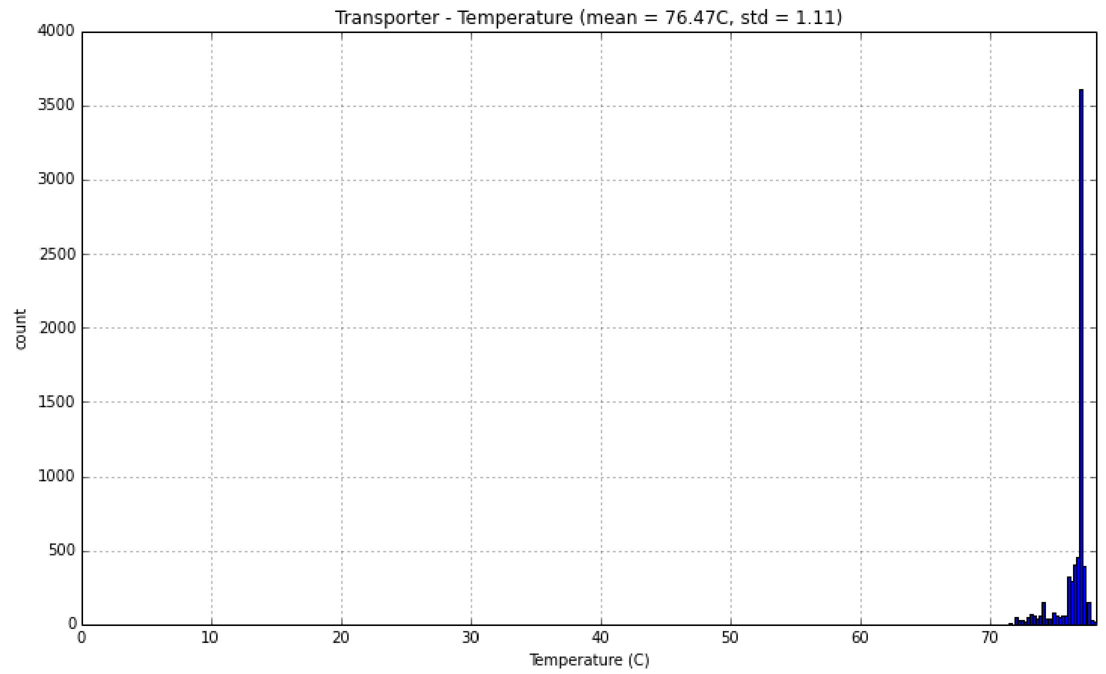






IPA Tuning flow (9)

- The temperature histogram shows the same information as the previous temperature plot but as a histogram.
- It allows you to see the spread of temperatures and how well tuned the solution is.
- Note that for some benchmarks that have parts that are not CPU or GPU intensive (like AnTuTu), it's not possible (or desirable) to have the device always at control temperature.



Things to observe

Temperature…

- Rate of temperature rise (from ambient)
  - Too gentle – not enough power
  - Performance being capped below control?
  - Too fast – leading to unacceptable overshoot
  - Duration of high frequency / performance above control?
  - k_pu and sustainable power
- Overshoot above control
  - Brief period of overshoot can give turbo like behaviour
  - Check when all devices loaded – is the overshoot acceptable?
  - Affected by k_po, k_i
- Time for temperature to return to control temperature once crossed
  - Does it take too long?
  - k_i

Things to observe

Temperature

- Temperature not controlled with all devices at lowest power
  - Control temperature too low?
  - Other means of reducing power control required
  - Hotplug
  - Turn off shaders for GPU
  - Controlling other heat generating devices

Things to observe

Power

- Unweighted request follows expected workload pattern
- Output power decreases above control temperature
- Allocation of power follows weighted power requests

Other support & documentation

- Linux kernel level documentation:
  - Documentation/thermal/power_allocator.txt
  - Documentation/thermal/cpu-cooling-api.txt

- Collaboration with Linaro in PMWG

- Consulting with ARM support ‘landing team’

FAQs

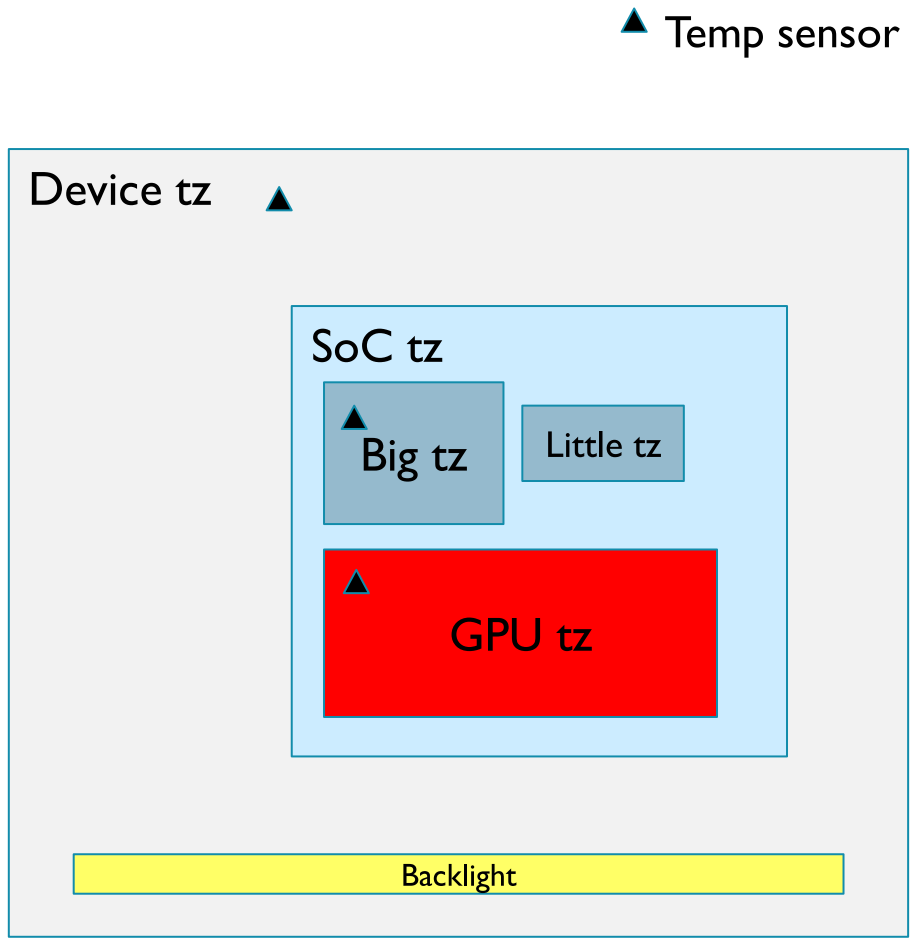
- Device vs SoC vs skin temperature control
  - Device thermal zone to prevent crossing junction temperatures
  - Use combination (max?) of sensors for SoC, if representative sensor not available
  - Looking into adding support to Linux
  - Skin temperature can be used to drive the PID loop as well
- Sensor resolution and accuracy
  - Linux thermal framework in milliCelsius
  - Higher resolution allows finer grained control of power
  - Lower resolution results in jumps in TDP estimate with corresponding bursty temperature and performance control
  - Might be mitigated by using a filter
- Accuracy of power models

Thermal zone hierarchy

- Thermal zones structured as tree (supports multiple temp sensors)
- Power restricted by cooling devices
- Allocated power is **minimum** of that requested by all thermal zones

⇒ Rapid limiting from CPU cluster temp sensor

⇒ Additional limiting from motherboard sensor



Thermal zone hierarchy (2)

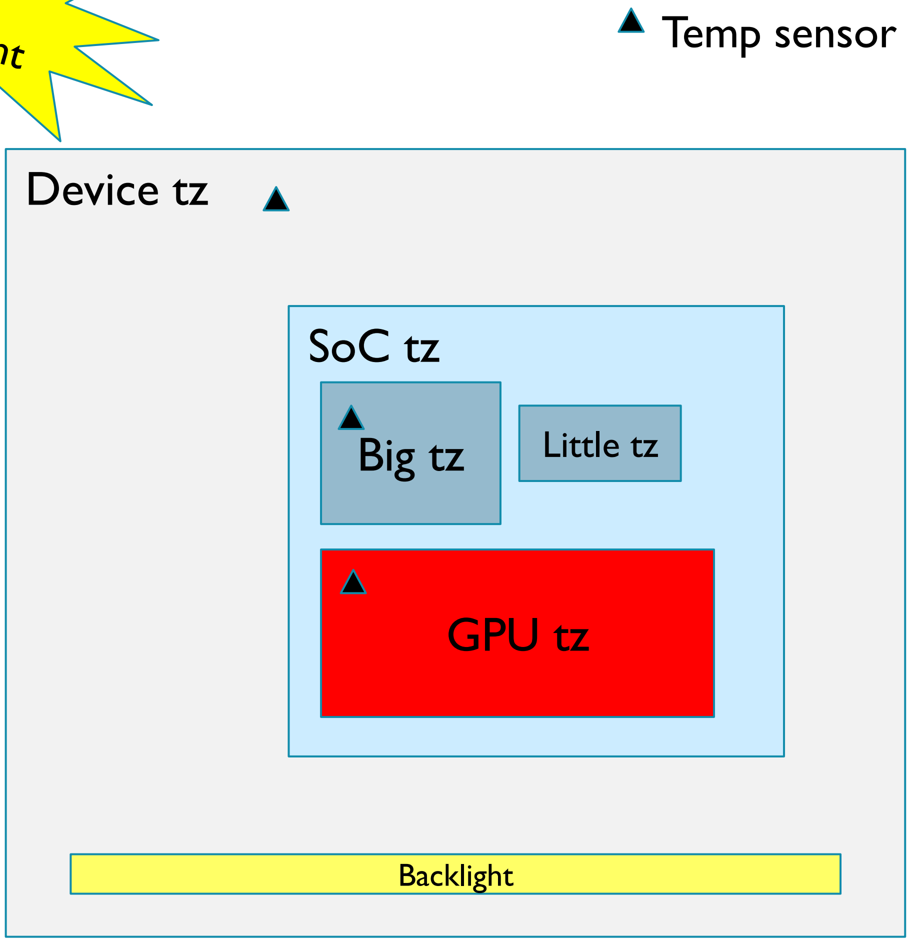
Under development

- In this example, the SoC thermal zone doesn't have a sensor for itself
- We can build its thermal zone by making the Big tz and GPU tz by children of it.
- With platform code

```
thermal_zone_add_subtz(soc_tz, big_tz);
thermal_zone_add_subtz(soc_tz, gpu_tz);
```

- With device tree

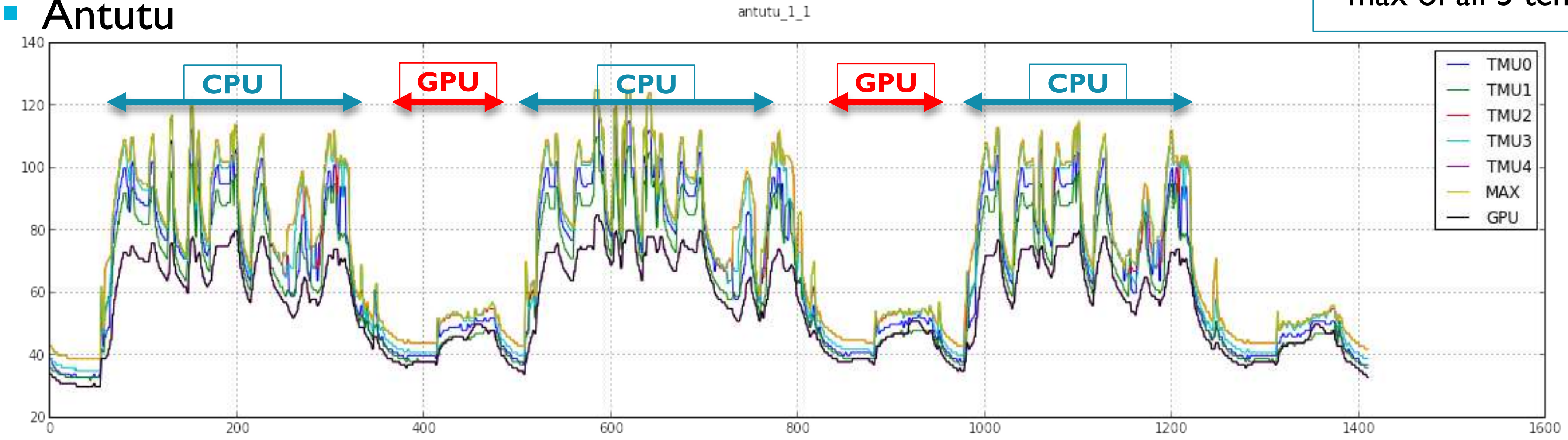
```
thermal-zones {
    soc_tz {
        polling-delay = <1000>;
        polling-delay-passive = <100>;
        sustainable_power = <2500>;
        thermal-sensors = <&big_tz &gpu_tz>;
    };
```



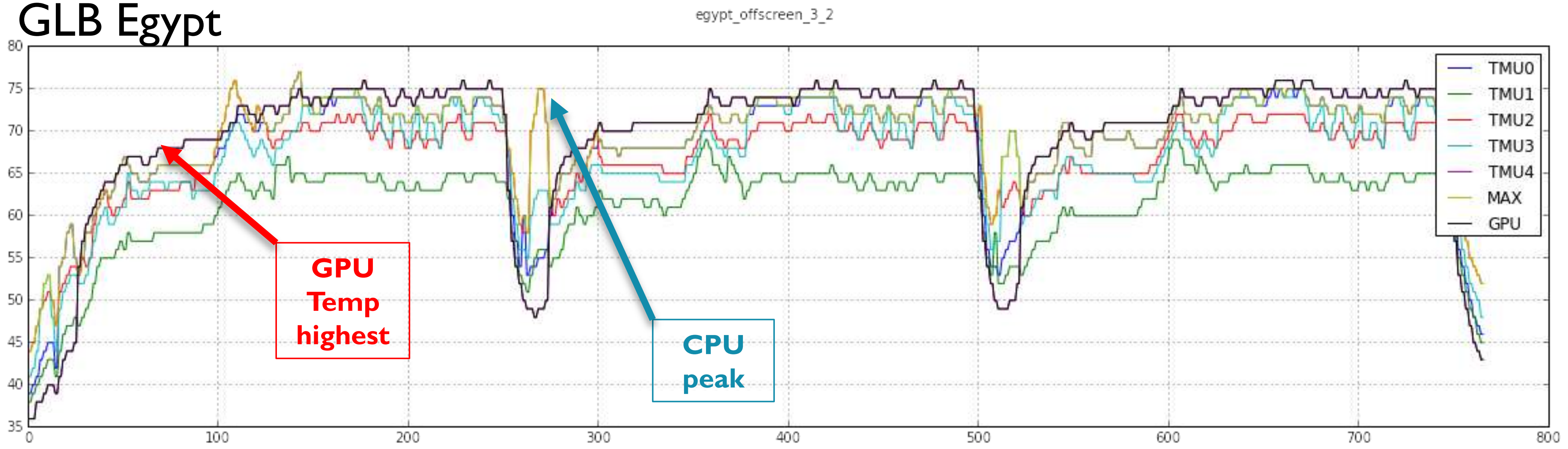
Multiple temp sensors

Combine multiple temp sensors – exynos port takes max of all 5 temp sensors

- Antutu



- GLB Egypt


egypt_offscreen_3_2
GPU Temp highest
CPU peak
TMU0
TMU1
TMU2
TMU3
TMU4
MAX
GPU