Linaro connect
Las Vegas 2016

Using tracing to tune and optimize EAS

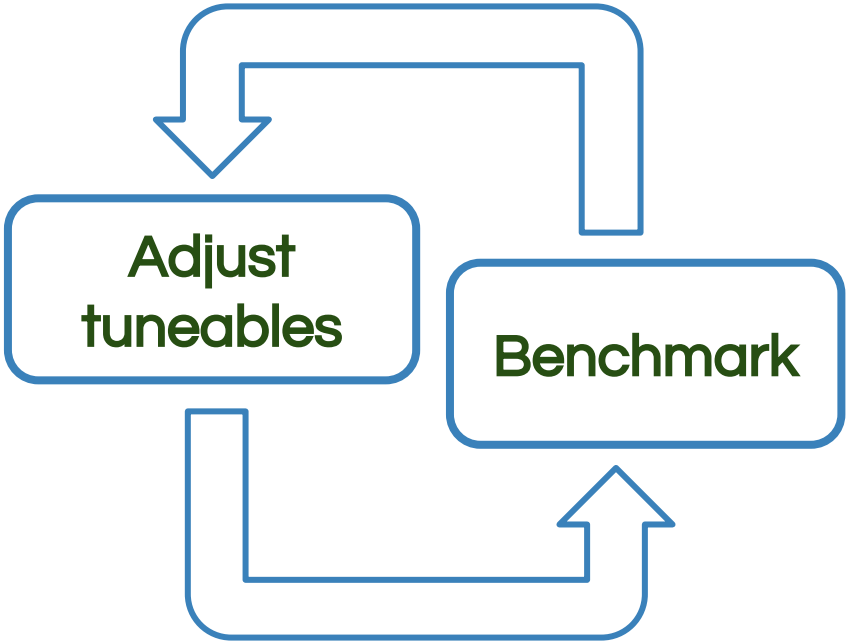
Leo Yan & Daniel Thompson
Linaro Support and Solutions Engineering

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Agenda

- Background
  - Review of typical workflow for GTS tuning
  - Introduce a workflow for EAS tuning
  - Quick introduction of the tools that support the new workflow
- Worked examples
  - Development platform for the worked examples
  - Analyze for task ping-pong issue
  - Analyze for small task staying on big core
- Further reading

Typical workflow for optimizing GTS



This simple workflow is easy to understand but has problems in practice.

Tunables are complex and interact with each other (making it hard to decide which tuneable to adjust).

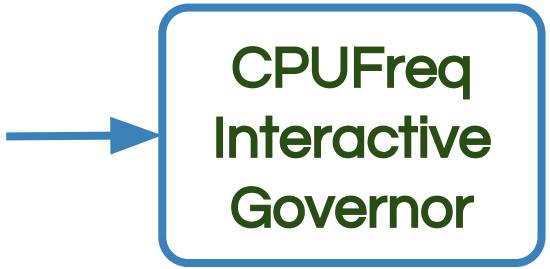
Tuning for multiple use-cases is difficult.

Tuning is SoC specific, optimizations will not necessarily apply to other SoCs.

GTS tunables

up_threshold
down_threshold
packing_enable
load_avg_period_ms
frequency_invariant_load_scale

hispeed_freq
go_hispeed_load
target_loads
Timer_rate
min_sample_time
above_hispeed_delay

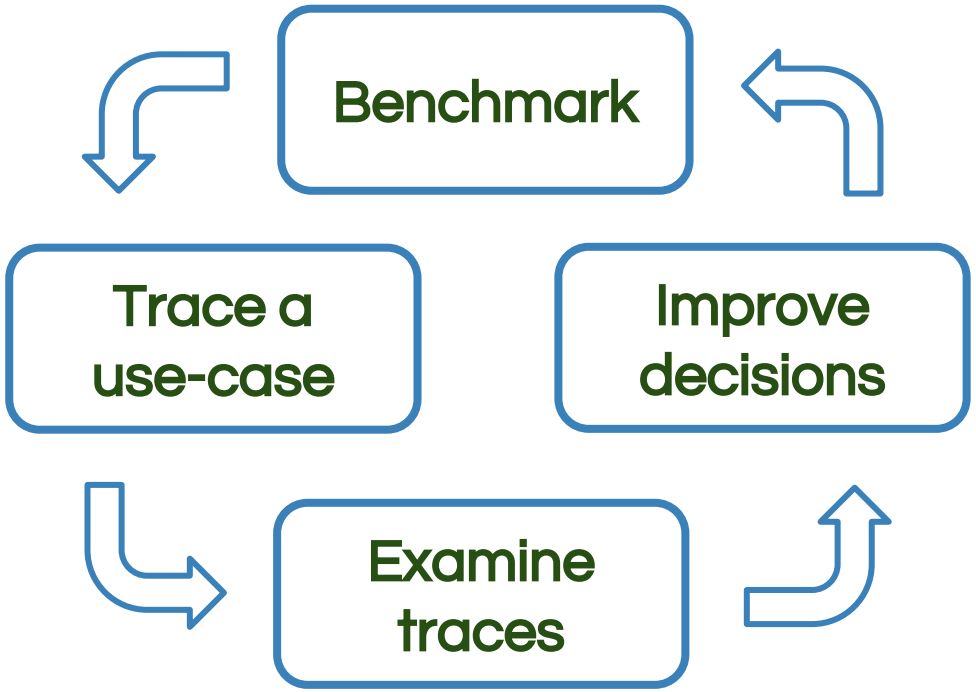


ENGINEERS
AND DEVICES
WORKING
TOGETHER

Agenda

- Background
  - Review of typical workflow for GTS tuning
  - Introduce a workflow for EAS tuning
  - Quick introduction of the tools that support the new workflow
- Worked examples
  - Development platform for the worked examples
  - Task ping-pong issue
  - Small task staying on big core
- Further reading

Typical workflow for optimizing EAS systems



Workflow is knowledge intensive.

Decisions can be improved by improving the power model or by finding new opportunities in the scheduler (a.k.a. debugging).

Optimizations are more portable.

- Can be shared for review
- Likely to benefit your new SoC

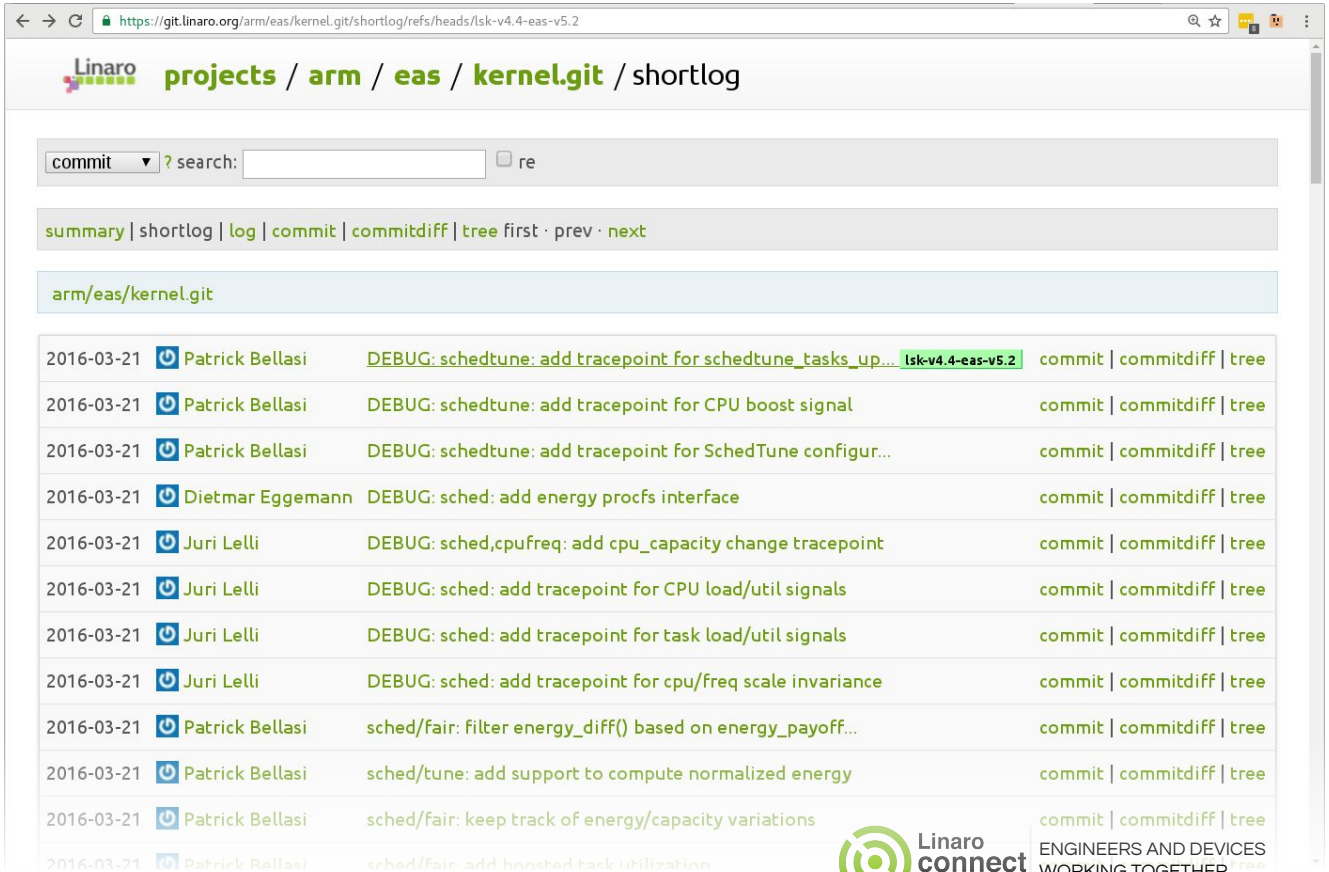
Trace points for EAS

Has a set of stock trace points in kernel for diving into debugging

Trace points are added by patches marked “DEBUG”

Not posted to LKML, currently only found in product focused patchsets

Enable kernel config: CONFIG_FTRACE



Trace points for EAS - cont.

Tracepoints in mainline kernel

PELT signals

sched_contrib_scale_f
sched_load_avg_task
sched_load_avg_cpu

Scheduler default events

sched_switch
sched_migrate_task
sched_wakeup
sched_wakeup_new

Tracepoints for EAS extension

EAS core

sched_energy_diff
sched_overutilized

SchedFreq

cpufreq_sched_throttled
cpufreq_sched_request_opp
cpufreq_sched_update_capacity

SchedTune

sched_tune_config
sched_boost_cpu
sched_tune_tasks_update
sched_tune_boostgroup_update
sched_boost_task
sched_tune_filter

LISA can be easily extended to support these trace points

E.g. enable trace points:

trace-cmd start -e sched_energy_diff -e sched_wakeup

Summary

Features	EAS	GTS
Make decision strategy	Power modeling	Heuristics thresholds
Frequency selection	Sched-freq or sched-util, integrated with scheduler	Governor's cascaded parameters
Scenario based tuning	schedTune (CGroup)	None

Energy awared scheduling (EAS) has very few tunables and thus requires a significantly different approach to tuning and optimization when compared to global task scheduling (GTS).

ENGINEERS
AND DEVICES
WORKING
TOGETHER

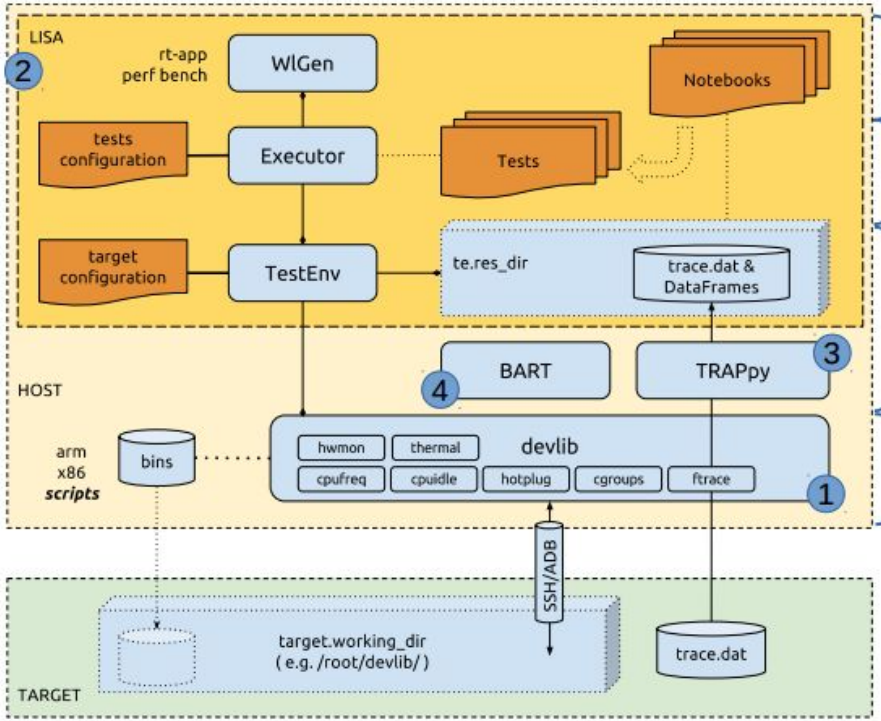
Agenda

- Background
  - Review of typical workflow for GTS tuning
  - Introduce a workflow for EAS tuning
  - Quick introduction of the tools that support the new workflow
- Worked examples
  - Development platform for the worked examples
  - Task ping-pong issue
  - Small task staying on big core
- Further reading

LISA - interactive analysis and testing

- “Distro” of python libraries for interactive analysis and automatic testing
- Libraries support includes
  - Target control and manipulation (set cpufreq mode, run this workload, initiate trace)
  - Gather power measurement data and calculate energy
  - Analyze and graph trace results
  - Test assertions about the trace results (e.g. big CPU does not run more than 20ms)
- Interactive analysis using ipython and jupyter
  - Provides a notebook framework similar to Maple, Mathematica or Sage
  - Notebooks mix together documentation with executable code fragments
  - Notebooks record the output of an interactive session
  - All permanent file storage is on the host
  - Trace files and graphs can be reexamined in the future without starting the target
- Automatic testing
  - Notebooks containing assertion based tests that can be converted to normal python

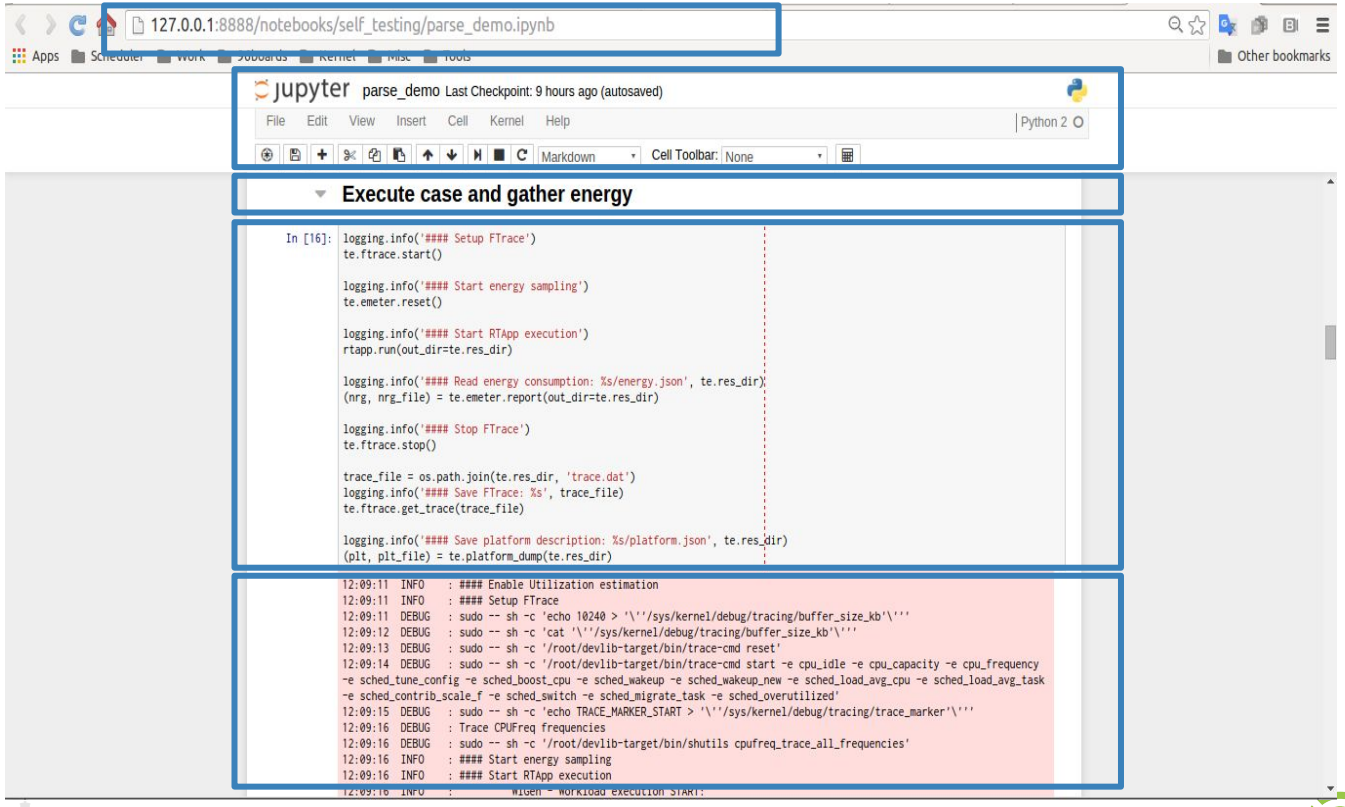
General workflow for LISA



https://github.com/ARM-software/devlib [1]
https://github.com/ARM-software/lisa [2]
https://github.com/ARM-software/trappy [3]
https://github.com/ARM-software/bart [4]

http://events.linuxfoundation.org/sites/events/files/slides/ELC16_LISA_20160326.pdf

LISA interactive test mode



http://127.0.0.1:8888 with ipython file

Menu & control buttons

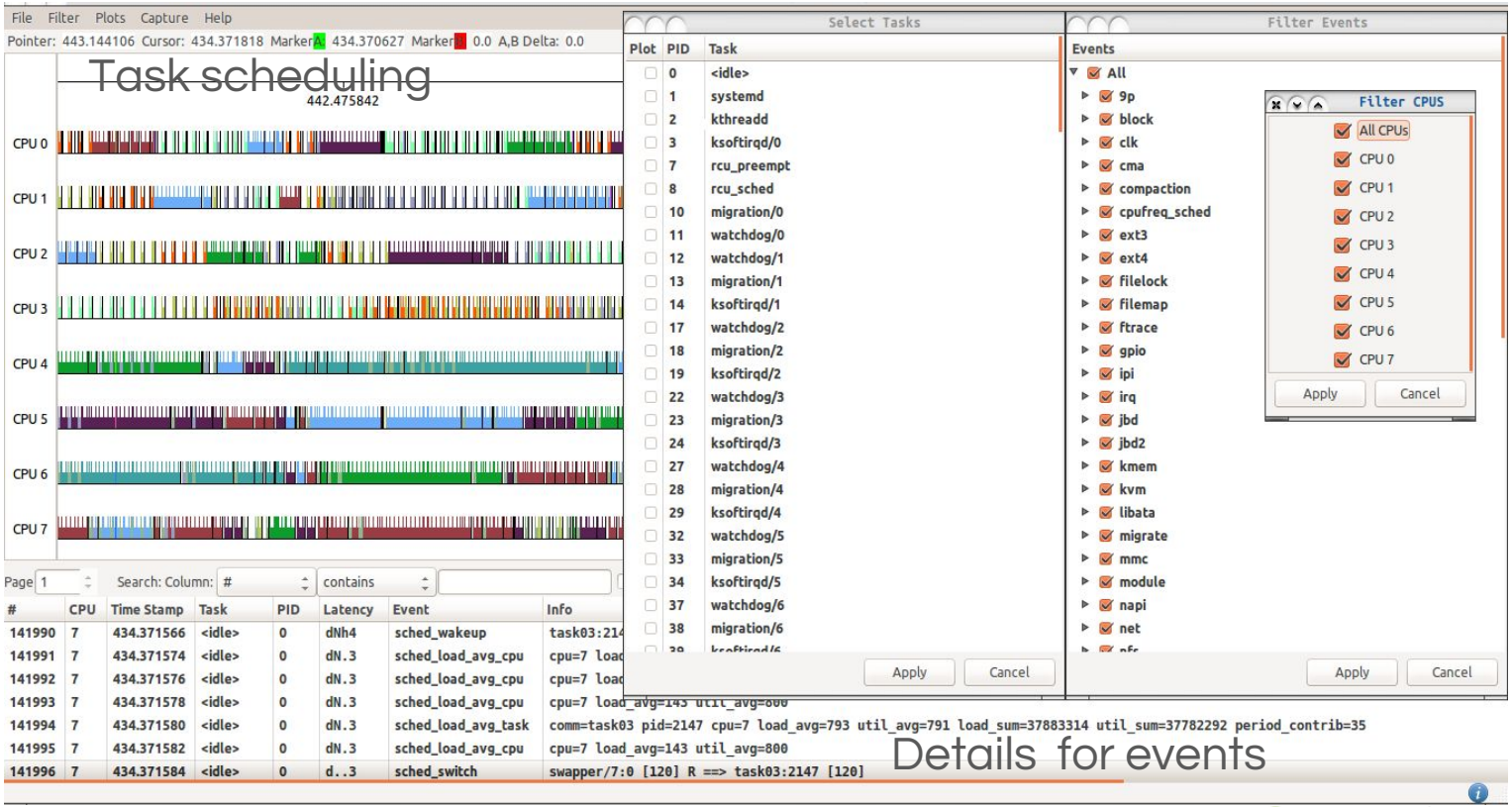
Markdown (headers)

Execute box with python programming

Result box, the results of experiments are recorded when next time reopen this file

kernelshark

Filters for events, tasks, and CPUs



ENGINEERS
AND DEVICES
WORKING
TOGETHER

Agenda

- Background
  - Review of typical workflow for GTS tuning
  - Introduce a workflow for EAS tuning
  - Quick introduction of the tools that support the new workflow
- Worked examples
  - Development platform for the worked examples
  - Task ping-pong issue
  - Small task staying on big core
- Further reading

Development platform for the worked examples

- All examples use artificial workloads to provoke a specific behaviour
  - It turned out to be quite difficult to deliberately provoke undesired behavior!
- Examples are reproducible on 96Boards HiKey
  - Octo-A53 multi-cluster (2x4) SMP device with five OPPs per cluster
    - Not big.LITTLE, and not using a fast/slow silicon process
  - We are able to fake a fast/slow system by using asymmetric power modeling parameters and artificially reducing the running/runnable delta time for “fast” CPU so the metrics indicate that is has a higher performance
- Most plots shown in these slides are copied from a LISA notebook
  - Notebooks and trace files have been shared for use after training

ENGINEERS
AND DEVICES
WORKING
TOGETHER

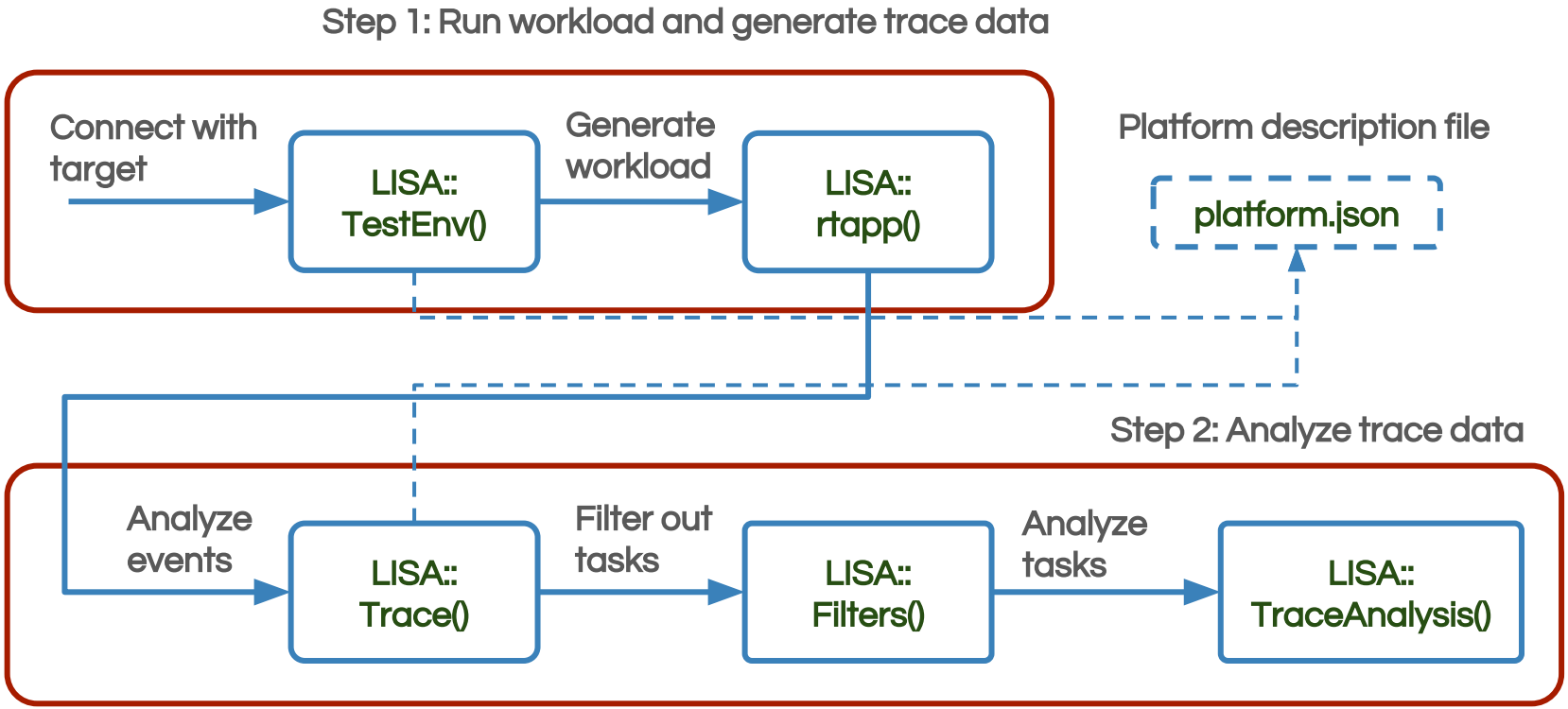
Agenda

- Background
  - Review of typical workflow for GTS tuning
  - Introduce a workflow for EAS tuning
  - Quick introduction of the tools that support the new workflow
- Worked examples
  - Development platform for the worked examples
  - Task ping-pong issue
  - Small task staying on big core
- Further reading

Testing environment

- CPU capacity info
  - The little core's highest capacity is 447@850MHz
  - The big core's highest capacity is 1024@1.1GHz
  - This case is running with correct power model parameters
- Test case
  - 16 small tasks are running with 15% utilization of little CPU (util ~= 67)
  - A single large task is running with 40% utilization of little CPU (util ~= 180)

General analysis steps



Connect with target board

In [ ]:
```
my_target_conf = {

    "platform"    : 'linux',
    "board"       : 'hikey',
    "modules"     : [
        'cpufreq'
    ],

    "host"        : '192.168.0.20',
    "username"    : 'root',
    "password"    : 'root',

    "rtapp-calib" : {
        "0": 259, "1": 250, "2": 253, "3": 251, "4": 251, "5": 251, "6": 252, "7": 250
    }
}

my_tests_conf = {

    "tools"   : ['rt-app', 'taskset', 'trace-cmd'],

    "ftrace"  : {
        "events" : [
            "sched_load_avg_cpu",
            "sched_load_avg_task",
        ],
        "buffsize" : 40960
    },

}
```

Specify target board info for connection

Calibration for CPUs

Tools copied to target board

Enable ftrace events

In [ ]:
```
te = TestEnv(target_conf=my_target_conf, test_conf=my_tests_conf)
target = te.target
```

Create connection

Generate and execute workload

```
rtapp = RTA(target, 'simple', calibration=te.calibration())

heavy  = Periodic(duty_cycle_pct=40, duration_s=5, period_ms=5)
light  = Periodic(duty_cycle_pct=15, duration_s=5, period_ms=5)

rtapp.conf(
    kind='profile',

    params={
        'task010': light.get(),
        'task011': light.get(),
        'task012': light.get(),
        'task013': light.get(),
        'task014': light.get(),
        'task015': light.get(),
        'task016': light.get(),
        'task017': light.get(),
        'task018': light.get(),
        'task019': light.get(),
        'task020': light.get(),
        'task021': light.get(),
        'task022': light.get(),
        'task023': light.get(),
        'task024': light.get(),
        'task025': light.get(),

        'task01': heavy.get(),

        run_dir=target.working_directory
);
```

Define workload

```
logging.info('#### Setup FTrace')
te.ftrace.start()

logging.info('#### Start energy sampling')
te.emeter.reset()

logging.info('#### Start RTApp execution')
rtapp.run(out_dir=te.res_dir)

logging.info('#### Read energy consumption: %s/energy.json', te.res_dir)
(nrg, nrg_file) = te.emeter.report(out_dir=te.res_dir)

logging.info('#### Stop FTrace')
te.ftrace.stop()

trace_file = os.path.join(te.res_dir, 'trace.dat')
logging.info('#### Save FTrace: %s', trace_file)
te.ftrace.get_trace(trace_file)

logging.info('#### Save platform description: %s/platform.json', te.res_dir)
(plt, plt_file) = te.platform_dump(te.res_dir)
```

Capture Ftrace data

Capture energy data

Execute workload

Graph showing task placement in LISA

In [5]:
```
trace_file = "/home/leoy/Work/disk/tools/x86/lisa-us/results/test_case_task_ping_pong/trace.dat"
print trace_file
events_to_parse = [
    "sched_wakeup",
    "sched_wakeup_new",
    "sched_load_avg_cpu",
    "sched_load_avg_task",
    "sched_boost_cpu",
    "sched_boost_task",
    "sched_energy_diff",
    "sched_switch",
    "sched_migrate_task",
    "sched_overutilized"]
ftrace = trappy.FTrace(trace_file, normalize_time=True, events=events_to_parse, window=(0, None))
```

Specify events to be extracted

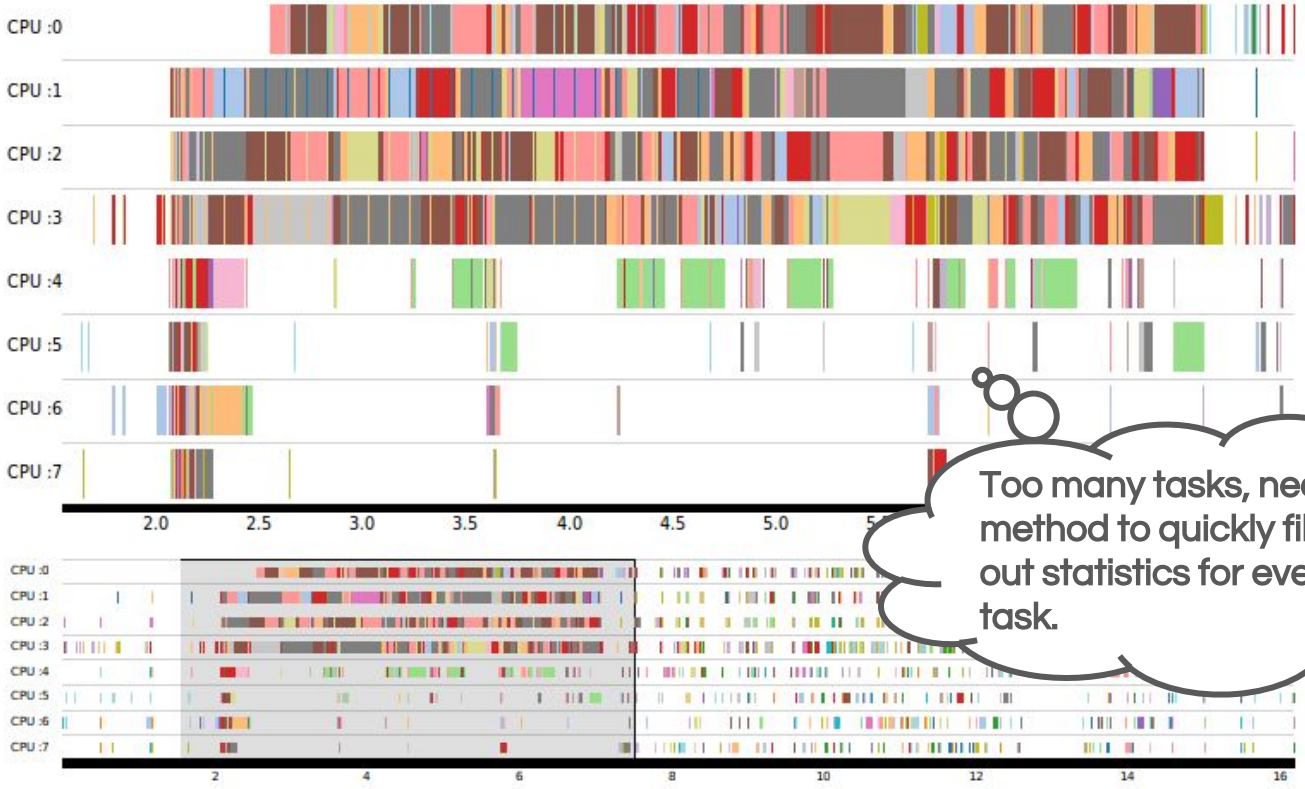
Specify time interval

```
/home/leoy/Work/disk/tools/x86/lisa-us/results/test_case_task_ping_pong/trace.dat
```

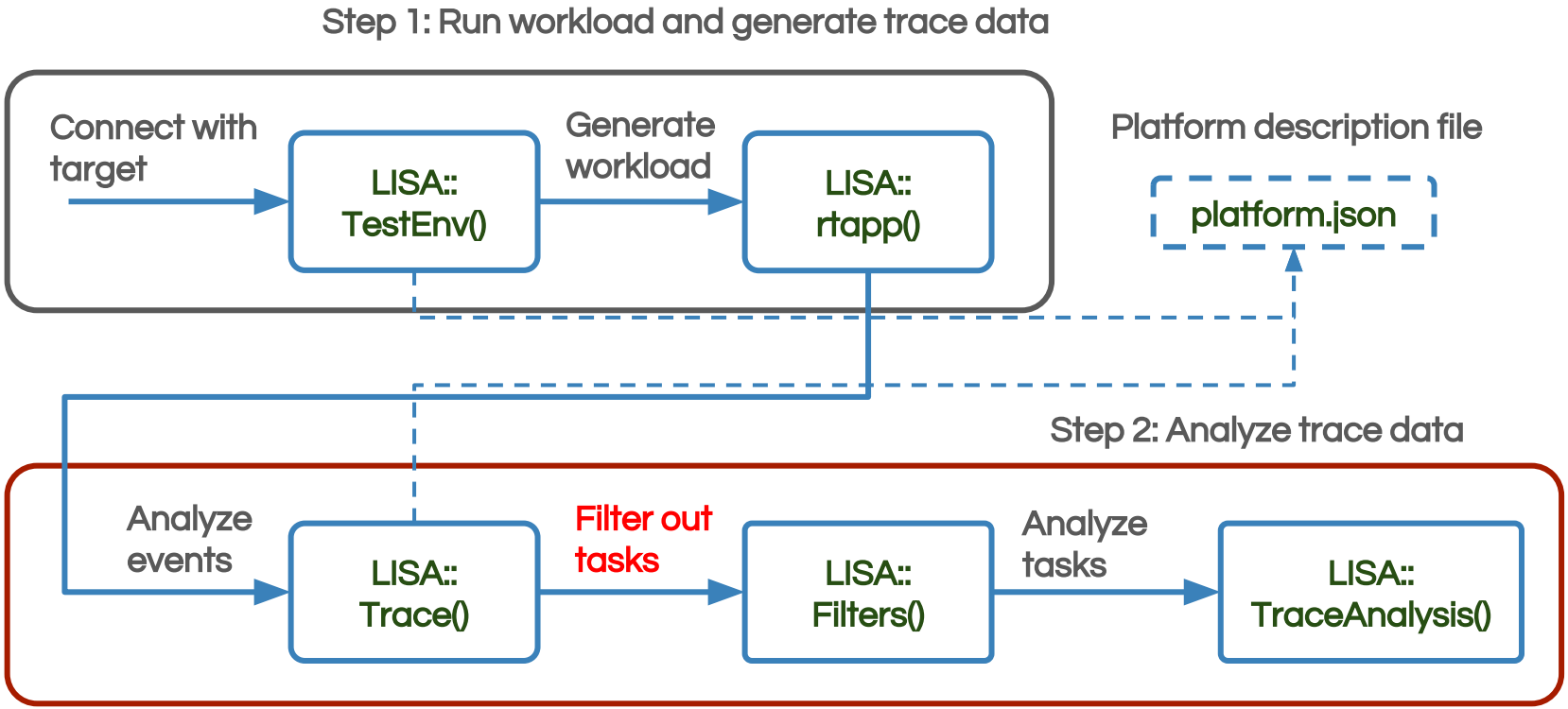
In [6]:
```
trappy.plotter.plot_trace(ftrace)
```

Display task placement graph

First make a quick graph showing task placement...



… and decide how to tackle step 2 analysis



Analyze trace data for events

```
events_to_parse = [
    "sched_switch",
    "sched_wakeup",
    "sched_wakeup_new",
    "sched_contrib_scale_f",
    "sched_load_avg_cpu",
    "sched_load_avg_task",
    "sched_tune_config",
    "sched_tune_tasks_update",
    "sched_tune_boostgroup_update",
    "sched_tune_filter",
    "sched_boost_cpu",
    "sched_boost_task",
    "sched_energy_diff",
    "cpu_frequency",
    "cpu_capacity",
    ""
]
```

```
# Load the LISA::Trace parsing module
from trace import Trace

(t_min, t_max) = (0, None)

trace = Trace(platform,
              tracefile,
              events_to_parse,
              trace_format="Ftrace",
              window=(t_min, t_max))
```

Format:
"SYSTRACE" or "Ftrace"

trace.dat

```
platform.json
{
    "clusters": {
        "big": [
            4,
            5,
            6,
            7
        ],
        "little": [
            0,
            1,
            2,
            3
        ]
    },
    "cpus_count": 8,
    "freqs": {
        "big": [
            208000,
            432000,
            729000,
            960000,
            1200000
        ],
        "little": [
            208000,
            432000,
            729000,
            960000,
            1200000
        ]
    },
    [...]
}
```

Selecting only task of interest (big tasks)

```
from filters import Filters

fl = Filters(trace)
fl.setXTimeRange(t_min, t_max)
```

```
# Get a list of tasks which are the most big in the trace
top_big_tasks = fl.topBigTasks(
    max_tasks=15,           # Maximum number of tasks to report
    min_utilization=10,     # Minimum utilization to be considered "big"
                            # default: LITTLE CPUs max capacity
    min_samples=100,        # Number of samples over the minimum utilization
)
```

```
top_big_tasks
{'mmcqd/0': 733,
 'Task01' : 2441,
 'task010': 2442,
 'task011': 2443,
 'task012': 2444,
 'task015': 2447,
 'task016': 2448,
 'task017': 2449,
 'task019': 2451,
 'task020': 2452,
 'task021': 2453,
 'task022': 2454,
 'task023': 2455,
 'task024': 2456,
 'task025': 2457}
```

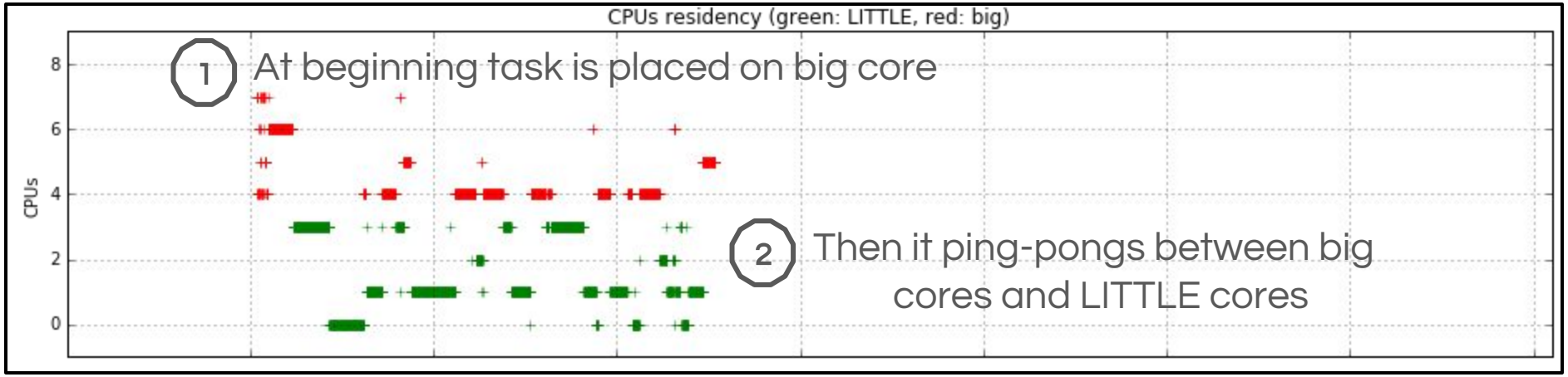
Plot big tasks with TraceAnalysis

```
from trace_analysis import TraceAnalysis

ta = TraceAnalysis(
    trace,                # LISA::Trace object
    tasks=top_big_tasks, # (optional) list of tasks to plot
    plotsdir=res_dir
)
```

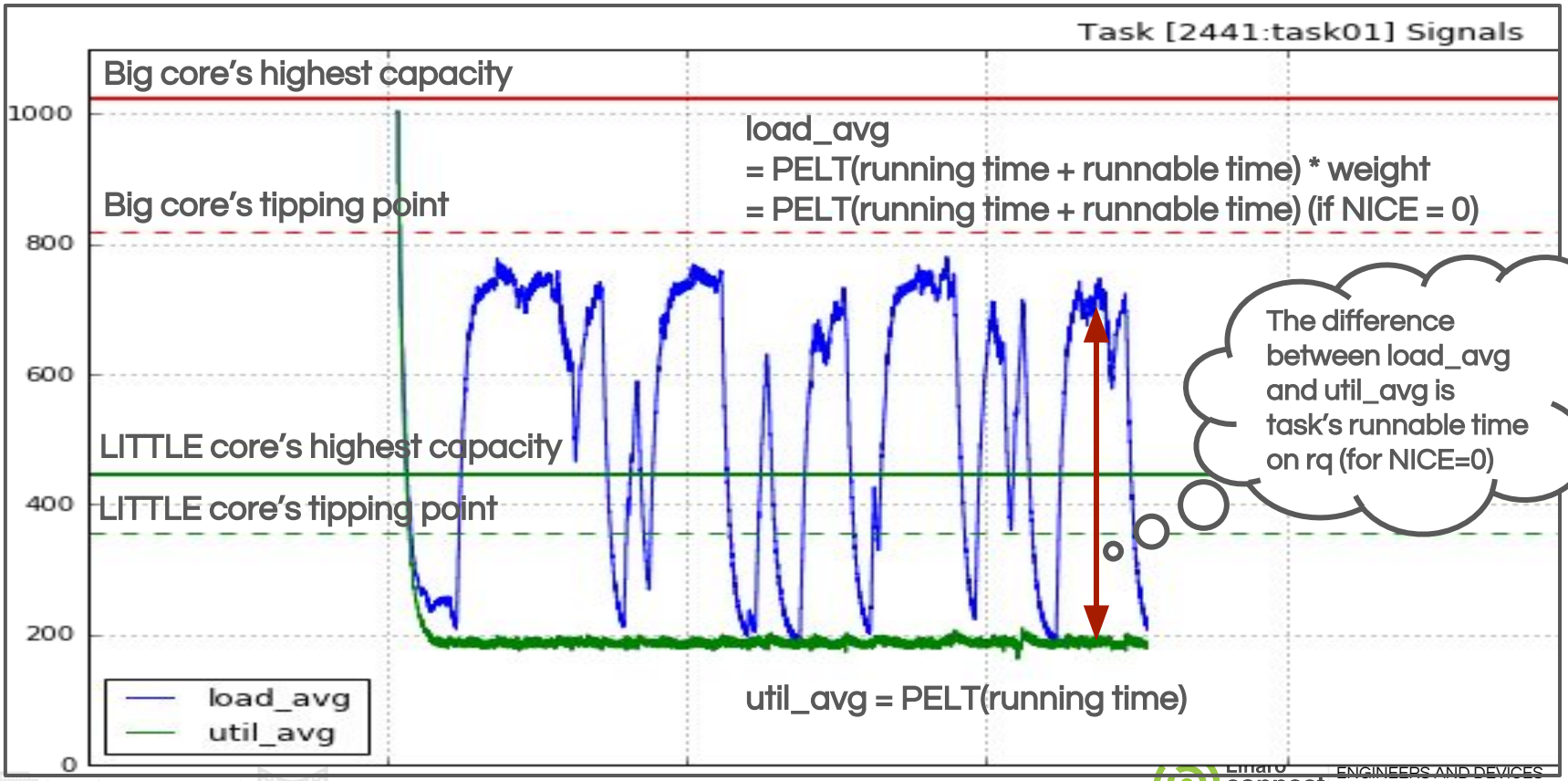
```
# Define time ranges for all the time based plots
ta.setXTimeRange(t_min, t_max)
```

```
ta.plotTasks(top_big_tasks)
```

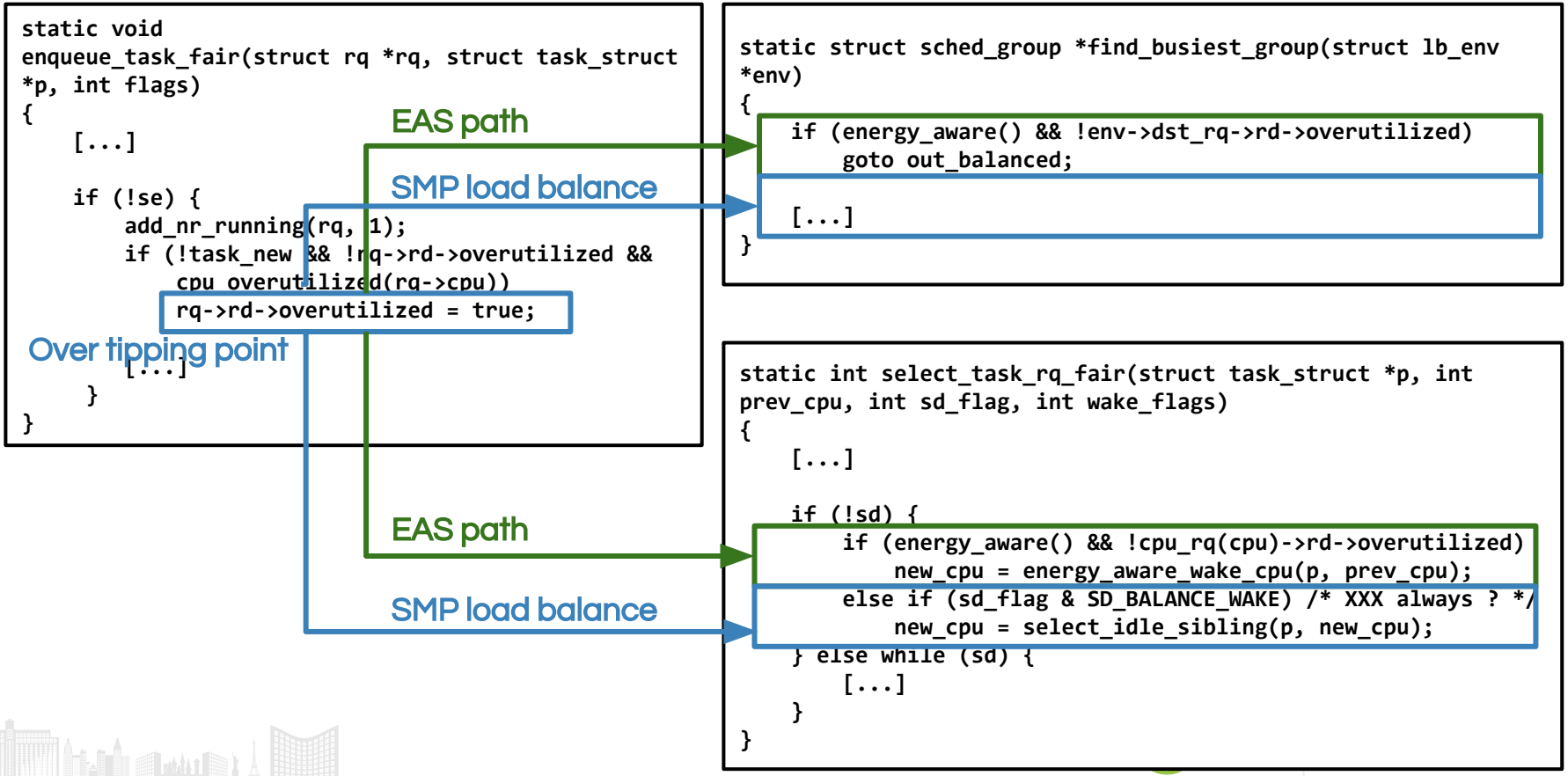
TraceAnalysis graph of task residency on CPUs



TraceAnalysis graph of task PELT signals



System cross tipping point for “over-utilized”

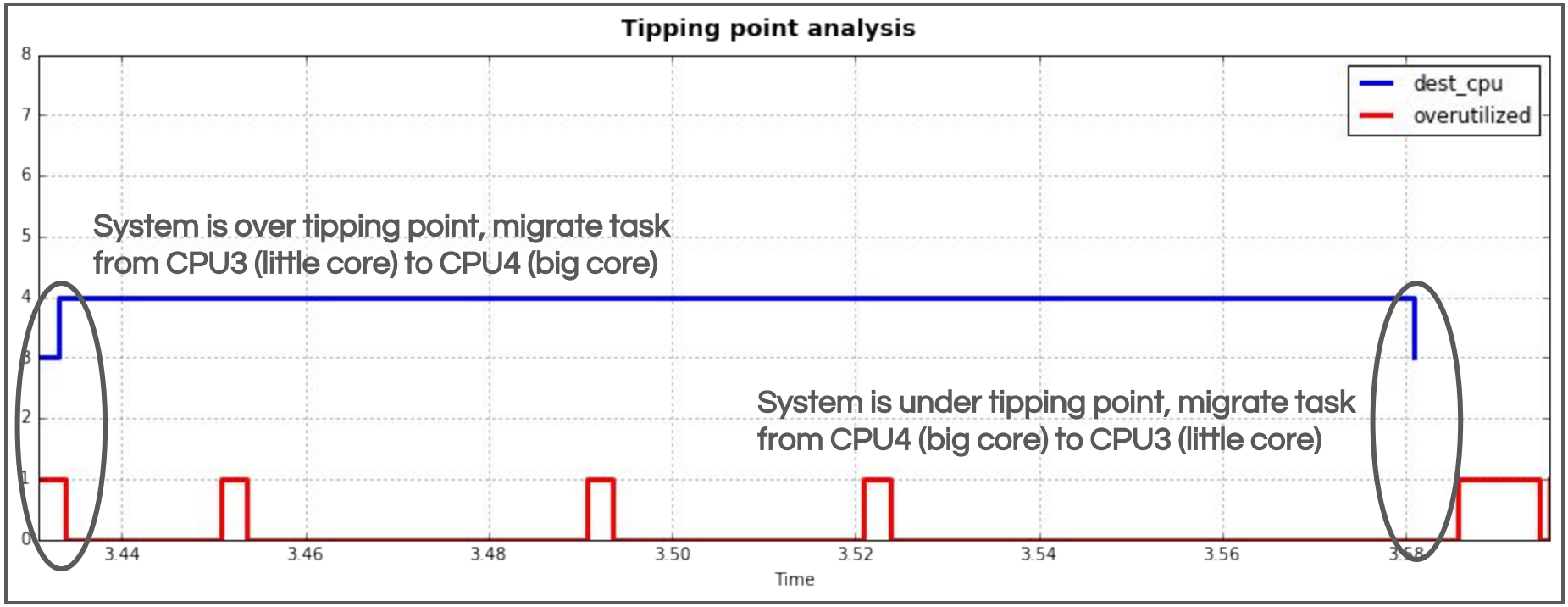


Write function to Analyze tipping point

If the LISA toolkit does not include the plotting function you need, you can write a plot function yourself

```
def analysis_tipping_point(pid):
    colors=['c', 'b', 'r']

    gs = gridspec.GridSpec(1, 1)
    fig = plt.figure(figsize=(16, 5))
    fig.subplots_adjust(top=0.92)
    plt.suptitle("Tipping point analysis", fontsize=14, fontweight='bold')

    axes = plt.subplot(gs[0, 0])

    dest_cpu = df_migrate_task[df_migrate_task.pid == pid][['dest_cpu']]
    dest_cpu.plot(ax=axes, drawstyle='steps-post', color=colors[1], linewidth=3, ylim=(0,8))

    tipping_point = df_overutilized[['overutilized']]
    tipping_point.plot(ax=axes, drawstyle='steps-post', color=colors[2], linewidth=3, ylim=(0,8))

    axes.grid(True)
```

Plot for tipping point



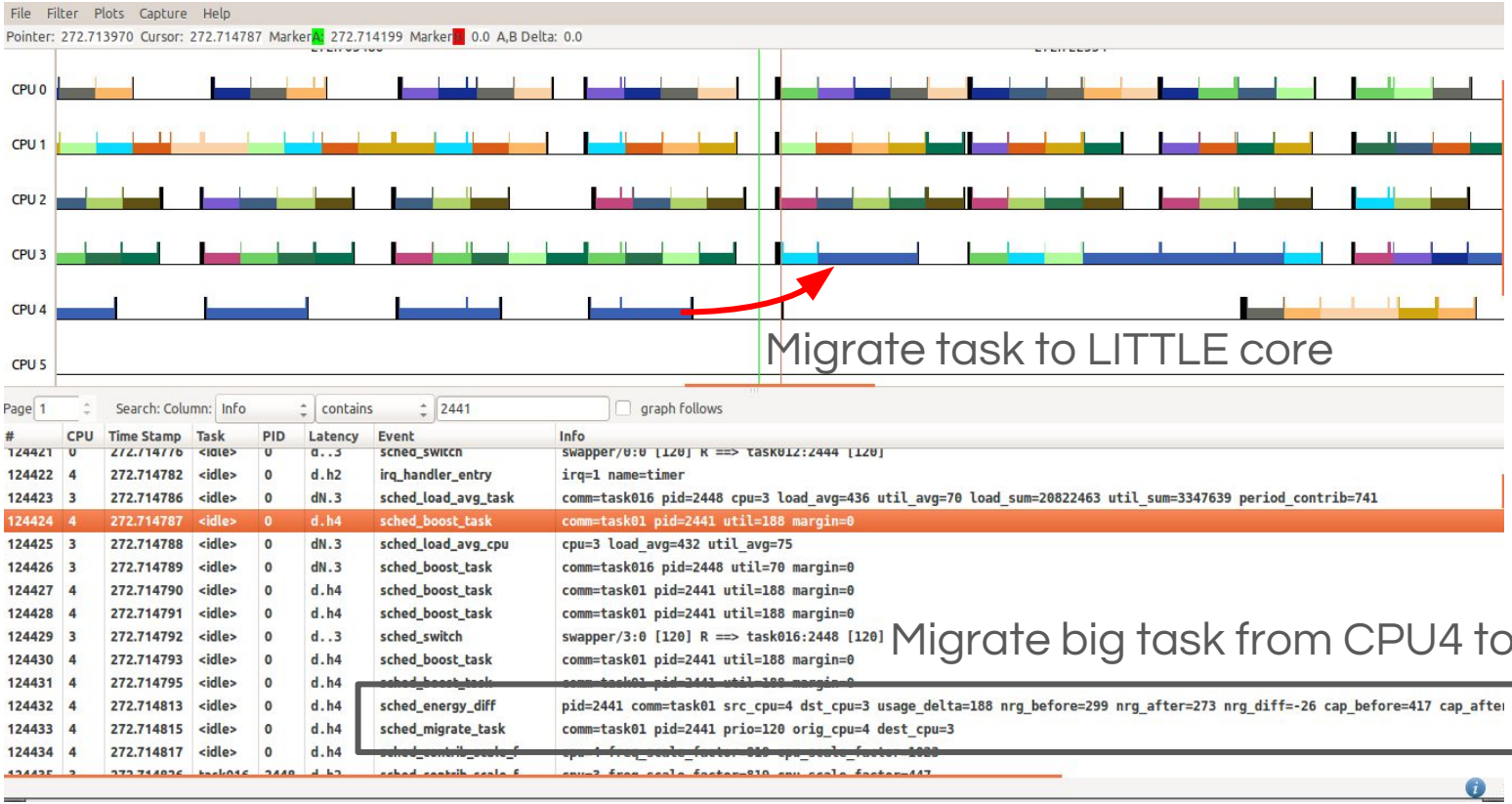
Detailed trace log for migration to big core

Page 1 Search: Column: Info contains 2441 graph follows

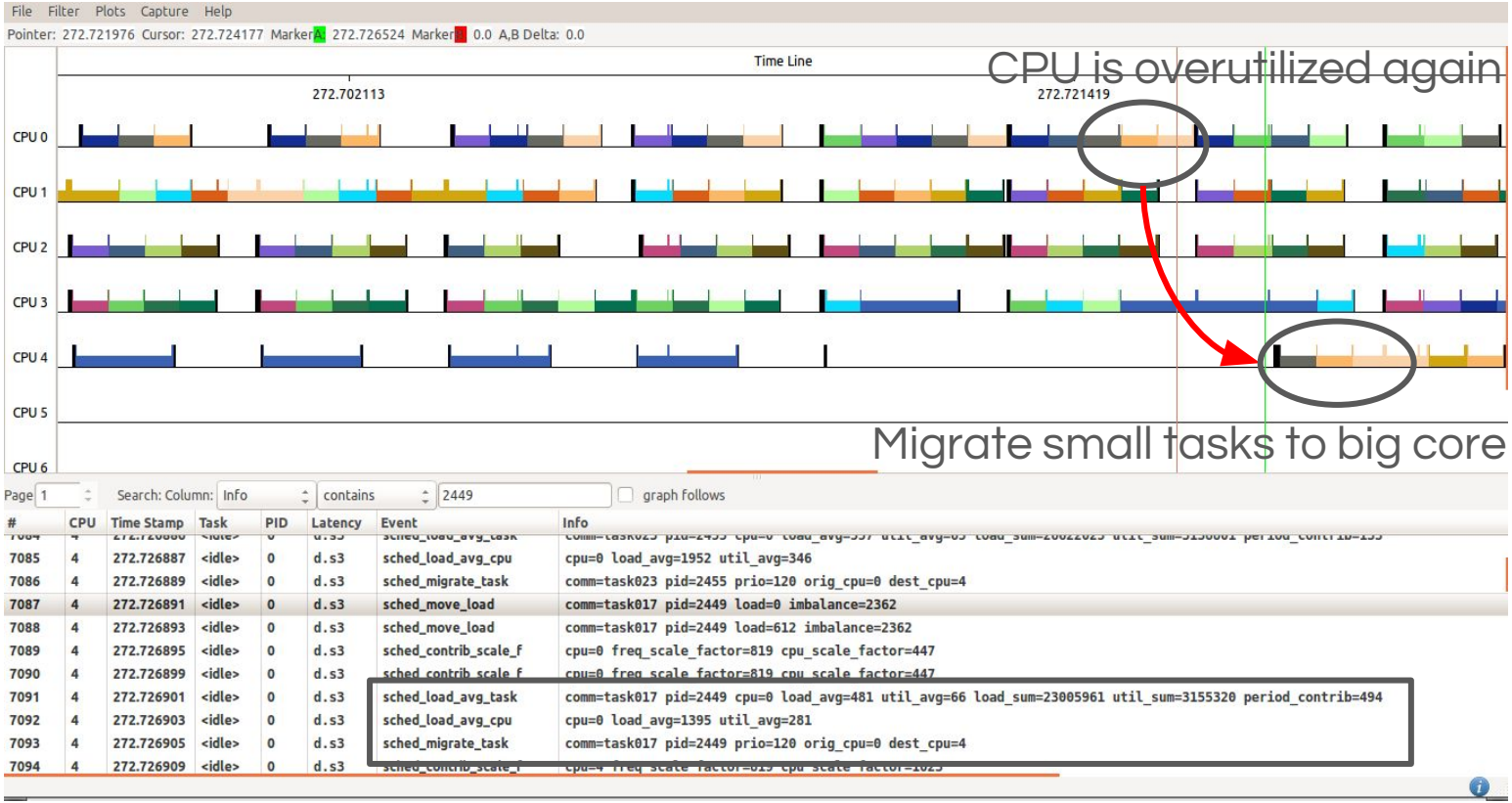
#	CPU	Time Stamp	Task	PID	Latency	Event	Info
113444	4	272.566984	<idle>	0	d.s3	sched_move_load	comm=task023 pid=2455 load=616 imbalance=3364
113445	4	272.566990	<idle>	0	d.s3	sched_contrib_scale_f	cpu=3 freq_scale_factor=819 cpu_scale_factor=447
113446	4	272.566993	<idle>	0	d.s3	sched_contrib_scale_f	cpu=3 freq_scale_factor=819 cpu_scale_factor=447
113447	4	272.566996	<idle>	0	d.s3	sched_load_avg_task	comm=task023 pid=2455 cpu=3 load_avg=629 util_avg=66 load_sum=30053516 util_sum=3172882 period_contrib=523
113448	4	272.566998	<idle>	0	d.s3	sched_load_avg_cpu	cpu=3 load_avg=2569 util_avg=456
113449	4	272.567003	<idle>	0	d.s3	sched_migrate_task	comm=task023 pid=2455 prio=120 orig_cpu=3 dest_cpu=4
113450	4	272.567006	<idle>	0	d.s3	sched_move_load	comm=task01 pid=2441 load=0 imbalance=2748
113451	4	272.567008	<idle>	0	d.s3	sched_move_load	comm=task01 pid=2441 load=933 imbalance=2748
113452	4	272.567012	<idle>	0	d.s3	sched_contrib_scale_f	cpu=3 freq_scale_factor=819 cpu_scale_factor=447
113453	4	272.567014	<idle>	0	d.s3	sched_contrib_scale_f	cpu=3 freq_scale_factor=819 cpu_scale_factor=447
113454	4	272.567016	<idle>	0	d.s3	sched_load_avg_task	comm=task01 pid=2441 cpu=3 load_avg=732 util_avg=180 load_sum=34974048 util_sum=8606844 period_contrib=468
113455	4	272.567018	<idle>	0	d.s3	sched_load_avg_cpu	cpu=3 load_avg=1940 util_avg=390
113456	4	272.567021	<idle>	0	d.s3	sched_migrate_task	comm=task01 pid=2441 prio=120 orig_cpu=3 dest_cpu=4
113457	4	272.567026	<idle>	0	d.s3	sched_contrib_scale_f	cpu=4 freq_scale_factor=819 cpu_scale_factor=1023

Migrate big task from CPU3 to CPU4

Issue 1: migration big task back to LITTLE core



Issue 2: migration small tasks to big core



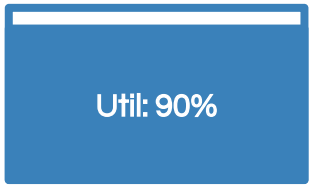
Tipping point criteria

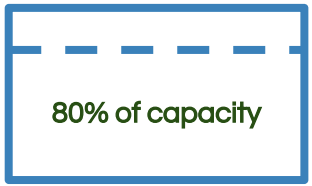
Over tipping point

Any CPU: cpu_util(cpu) > cpu_capacity(cpu) * 80%

E.g. LITTLE core:
cpu_capacity(cpu0) = 447

E.g. Big core:
cpu_capacity(cpu4) = 1024

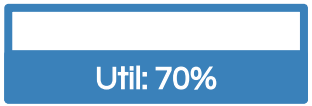




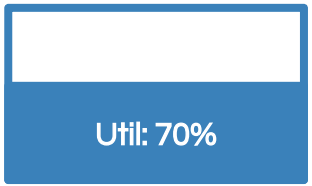
Under tipping point

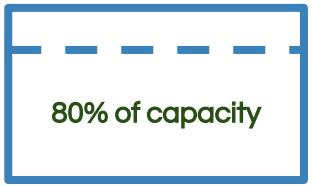
ALL CPUs: cpu_util(cpu) < cpu_capacity(cpu) * 80%

E.g. LITTLE core:
cpu_capacity(cpu0) = 447

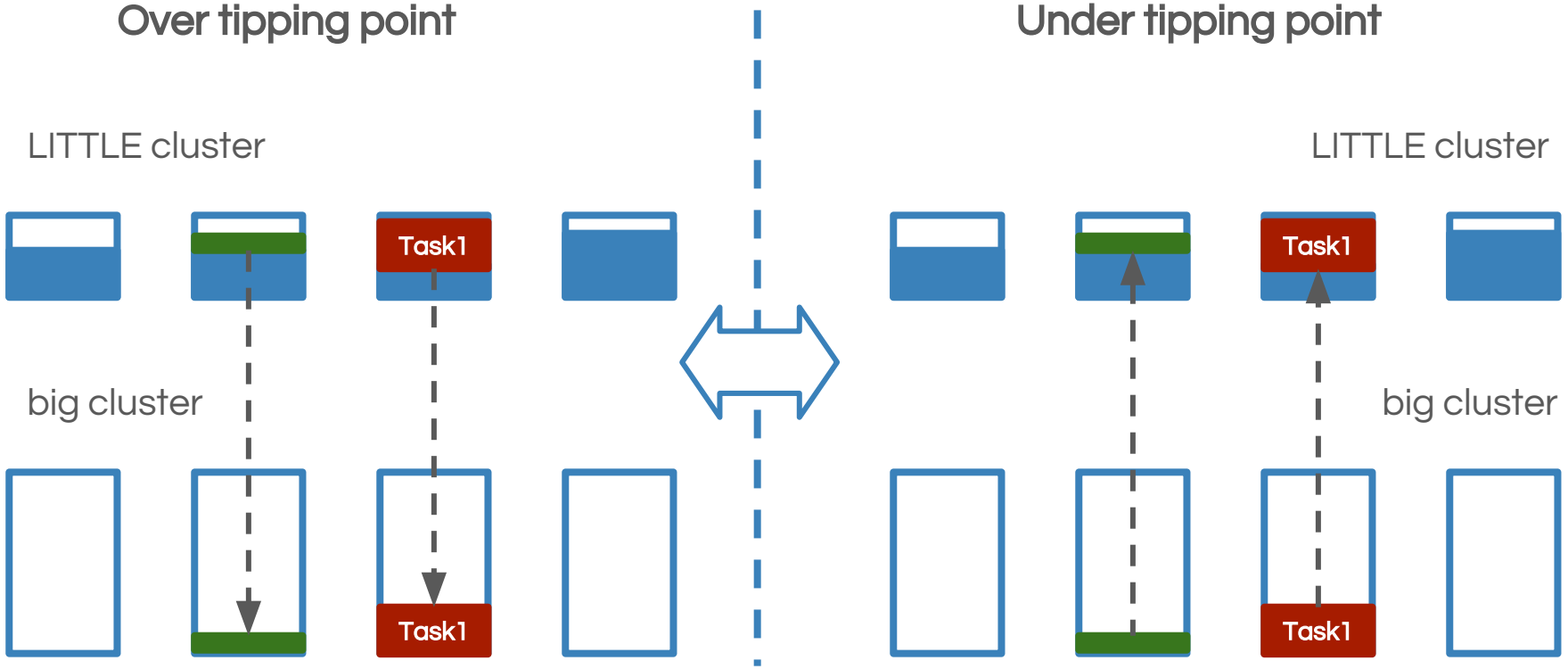


E.g. Big core:
cpu_capacity(cpu4) = 1024

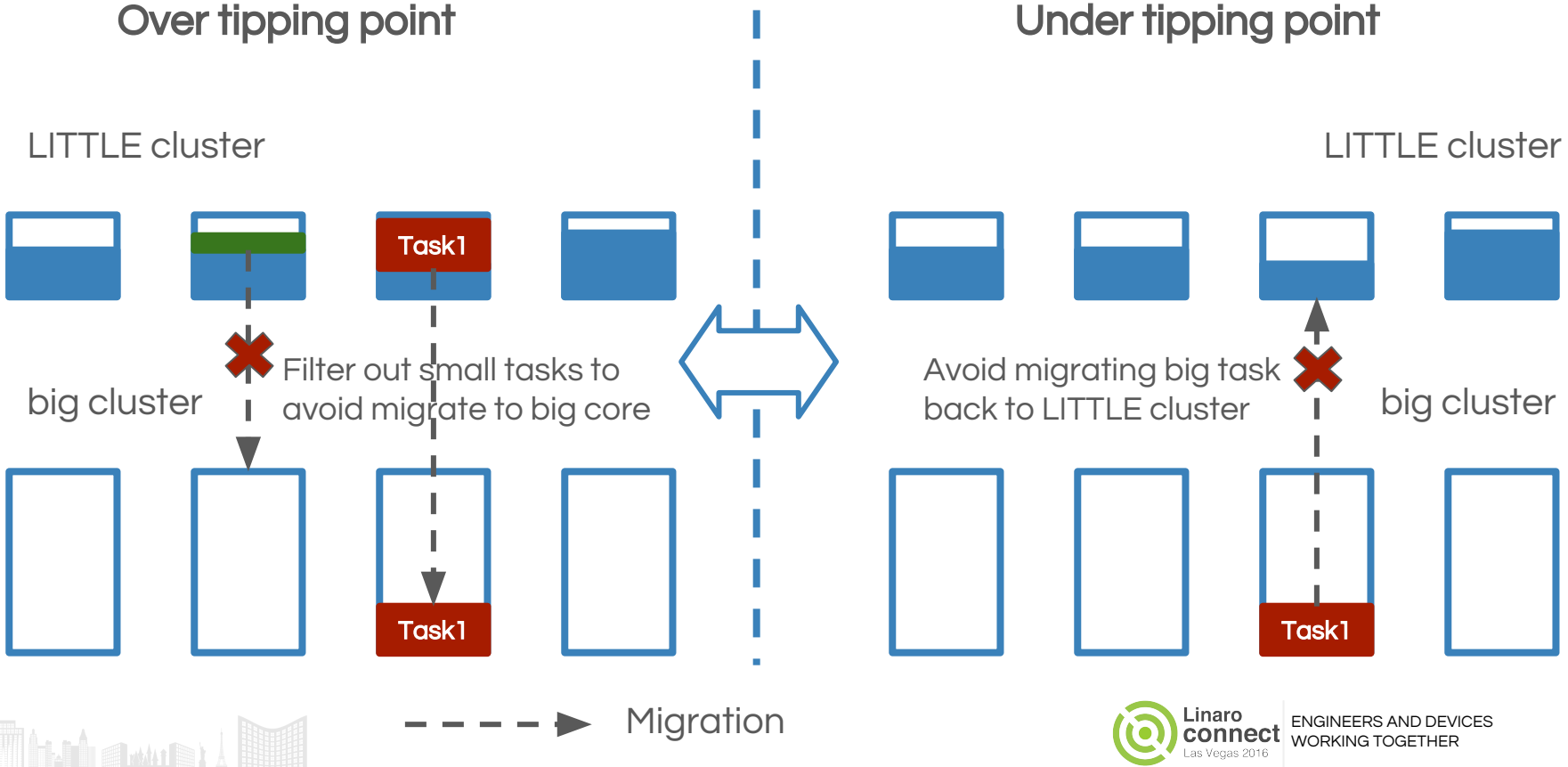




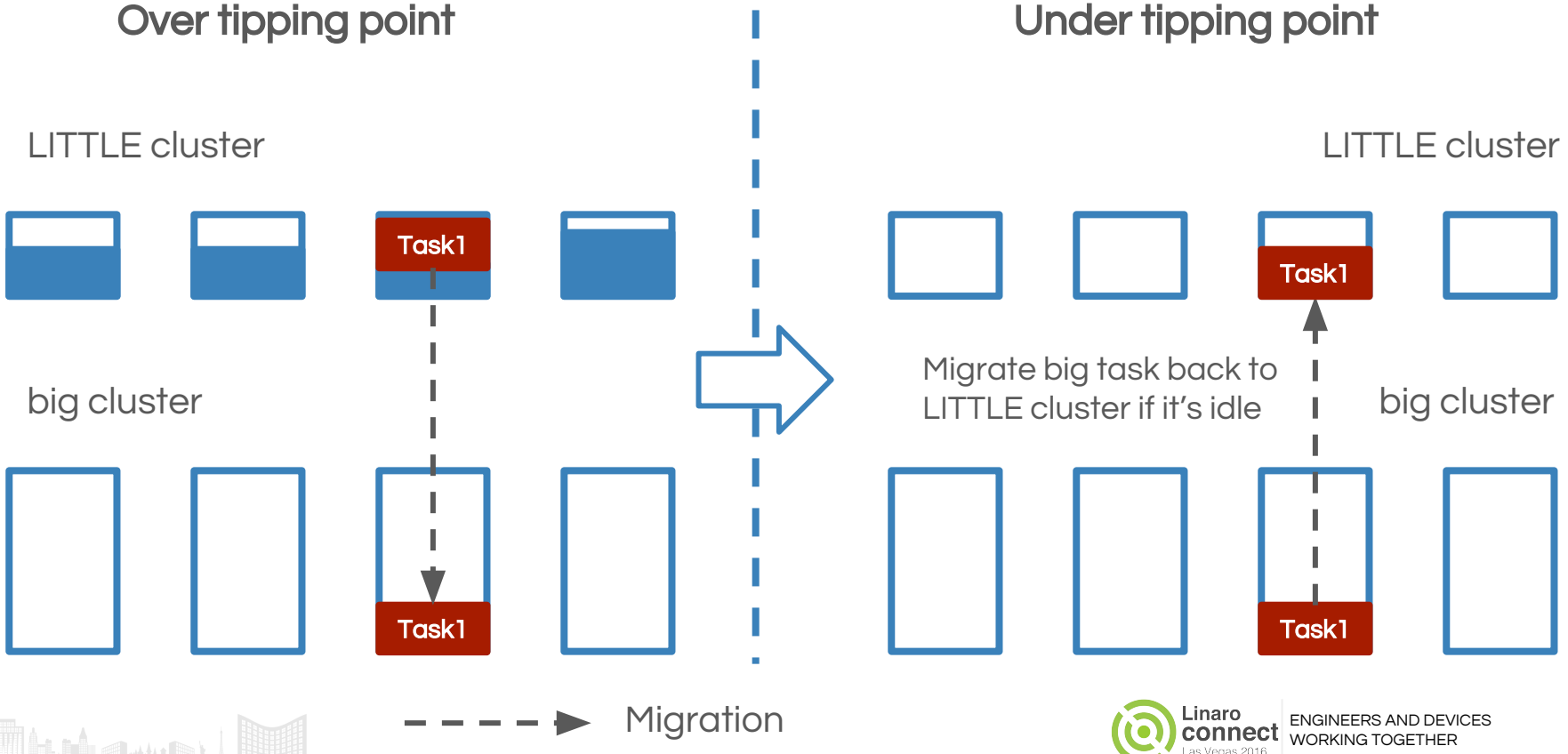
Phenomenon for ping-pong issue



Fixes for ping-pong issue



Fallback to LITTLE cluster after it is idle



Filter out small tasks for (tick, idle) load balance

```
static
int can_migrate_task(struct task_struct *p, struct lb_env *env)
{
        [...]

        if (energy_aware() &&
            (capacity_orig_of(env->dst_cpu) > capacity_orig_of(env->src_cpu))) {

                if (task_util(p) * 4 < capacity_orig_of(env->src_cpu))
                        return 0;

        }

        [...]
}
```

Filter out small tasks: task running time < ¼ LITTLE CPU capacity.
These tasks will NOT be migrated to big core after return 0.
Result: Only big tasks has a chance to migrate to big core.

Avoid migrating big task to LITTLE cluster

```
static int select_task_rq_fair(struct task_struct *p, int prev_cpu,
int sd_flag, int wake_flags)
{
    [...]

    if (!sd) {
        if (energy_aware() &&
            (!need_spread_task(cpu) || need_filter_task(p)))
            new_cpu = energy_aware_wake_cpu(p, prev_cpu);
        else if (sd_flag & SD_BALANCE_WAKE) /* XXX always ? */
            new_cpu = select_idle_sibling(p, new_cpu);
    } else while (sd) {
        [...]
    }
}
```

```
static bool need_spread_task(int cpu)
{
        struct sched_domain *sd;
        int spread = 0, i;

        if (cpu_rq(cpu)->rd->overutilized)
              return 1;

        sd = rcu_dereference_check_sched_domain(cpu_rq(cpu)->sd);
        if (!sd)
               return 0;

        for_each_cpu(i, sched_domain_span(sd)) {
               if (cpu_rq(i)->cfs.h_nr_running >= 1 &&
                  cpu_halfutilized(i)) {
                       spread = 1;
                       Break;
               }
        }

        return spread;
}
```

Check if cluster is busy or not as well as checking system tipping point:

- Easier to spread tasks within cluster if cluster is busy
- Fallback to migrating big task when cluster is idle

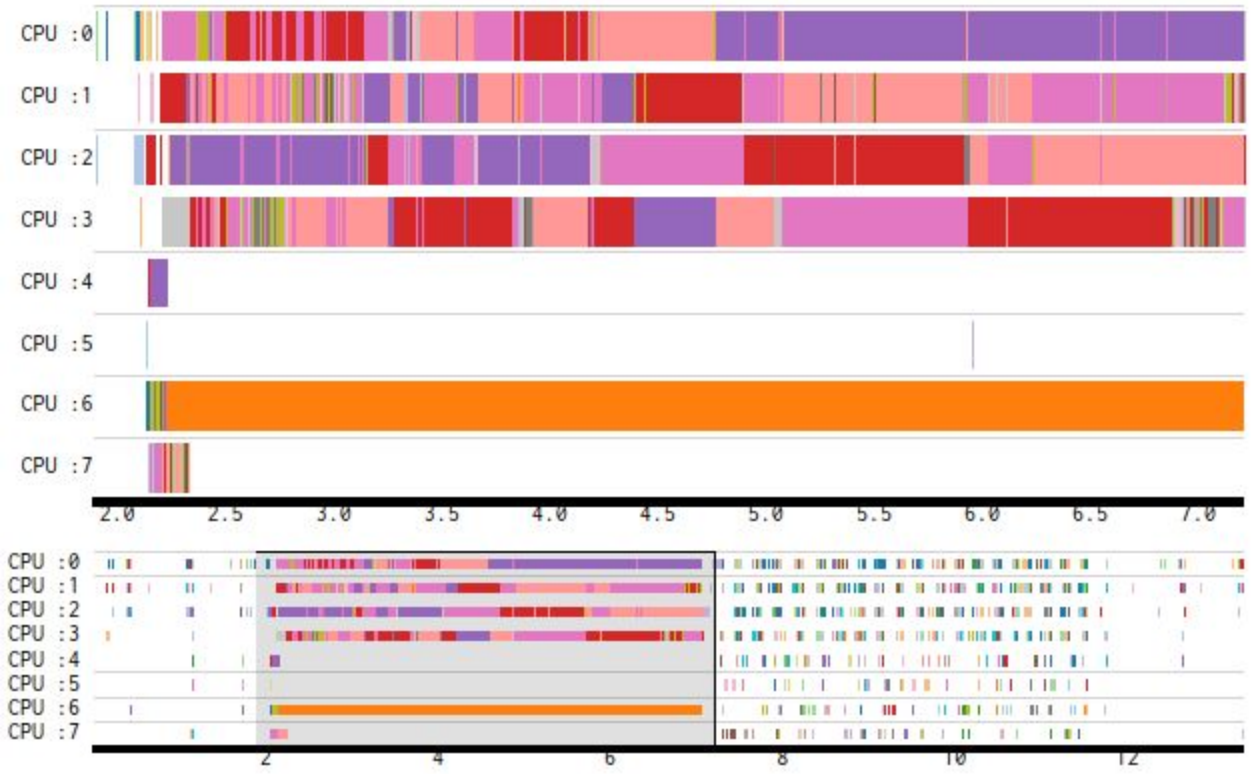
Filter out small tasks for wake up balance

```
static bool need_filter_task(struct task_struct *p)
{
        int cpu = task_cpu(p);
        int origin_max_cap = capacity_orig_of(cpu);
        int target_max_cap = cpu_rq(cpu)->rd->max_cpu_capacity.val;
        struct sched_domain *sd;
        struct sched_group *sg;

        sd = rcu_dereference(per_cpu(sd_ea, cpu));
        sg = sd->groups;
        do {
                int first_cpu = group_first_cpu(sg);

                if (capacity_orig_of(first_cpu) < target_max_cap &&
                    task_util(p) * 4 < capacity_orig_of(first_cpu))
                        target_max_cap = capacity_orig_of(first_cpu);

        } while (sg = sg->next, sg != sd->groups);

        if (target_max_cap < origin_max_cap)
                return 1;

        return 0;
}
```

Two purposes of this function:

- Select small tasks (task running time < ¼ LITTLE CPU capacity) and keep them on the energy aware path
- Prevent energy aware path for big tasks on the big core from doing harm to little tasks.

Results after applying patches



The big task always run on CPU6 and small tasks run on LITTLE cores!

ENGINEERS
AND DEVICES
WORKING
TOGETHER

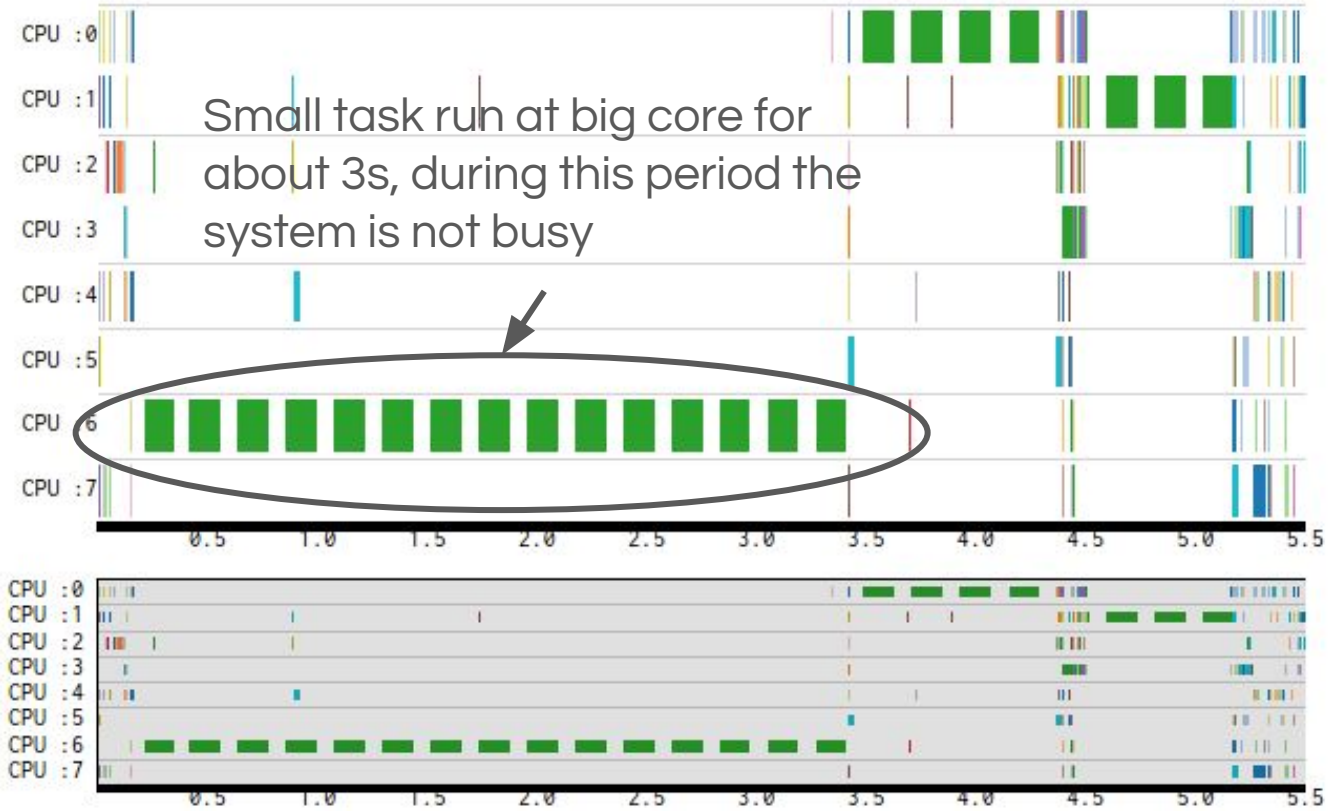
Agenda

- Background
  - Review of typical workflow for GTS tuning
  - Introduce a workflow for EAS tuning
  - Quick introduction of the tools that support the new workflow
- Worked examples
  - Development platform for the worked examples
  - Task ping-pong issue
  - Small task staying on big core
- Further reading

Testing environment

- Testing environment
  - The LITTLE core's highest capacity is 447@850MHz
  - The big core's highest capacity is 1024@1.1GHz
  - Single small task is running with 9% utilization of big CPU (util ~= 95)
- Phenomenon
  - The single small task runs on big CPU for long time, even though its utilization is well below the tipping point

Global View For Task's Placement



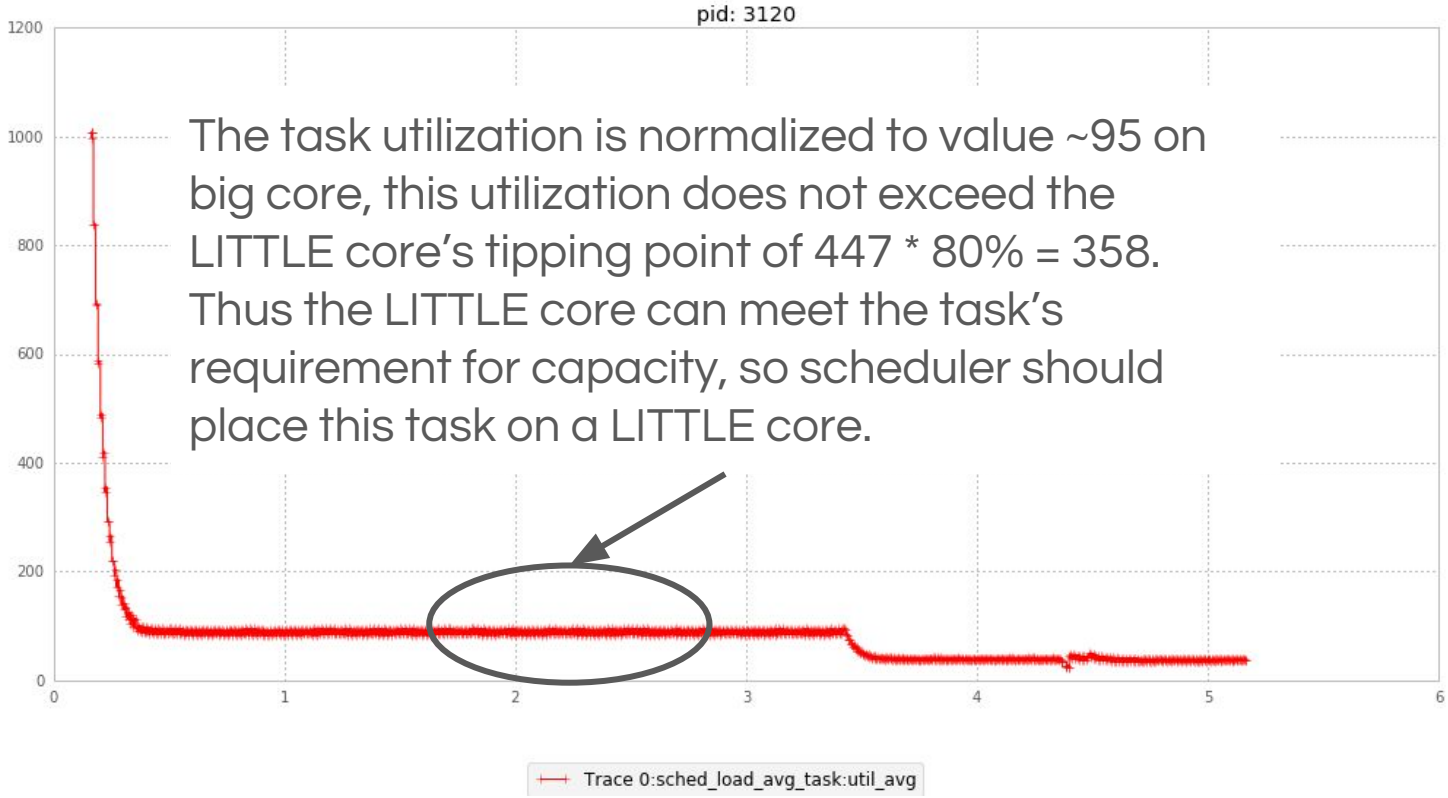
Analyze task utilization

```
trappy.LinePlot(
    ftrace,
    signals=[
        'sched_load_avg_task:util_avg'
    ],
    pivot='pid',
    filters={'comm': ['task1']},
    drawstyle='steps-post',
    marker = '+').view()
```

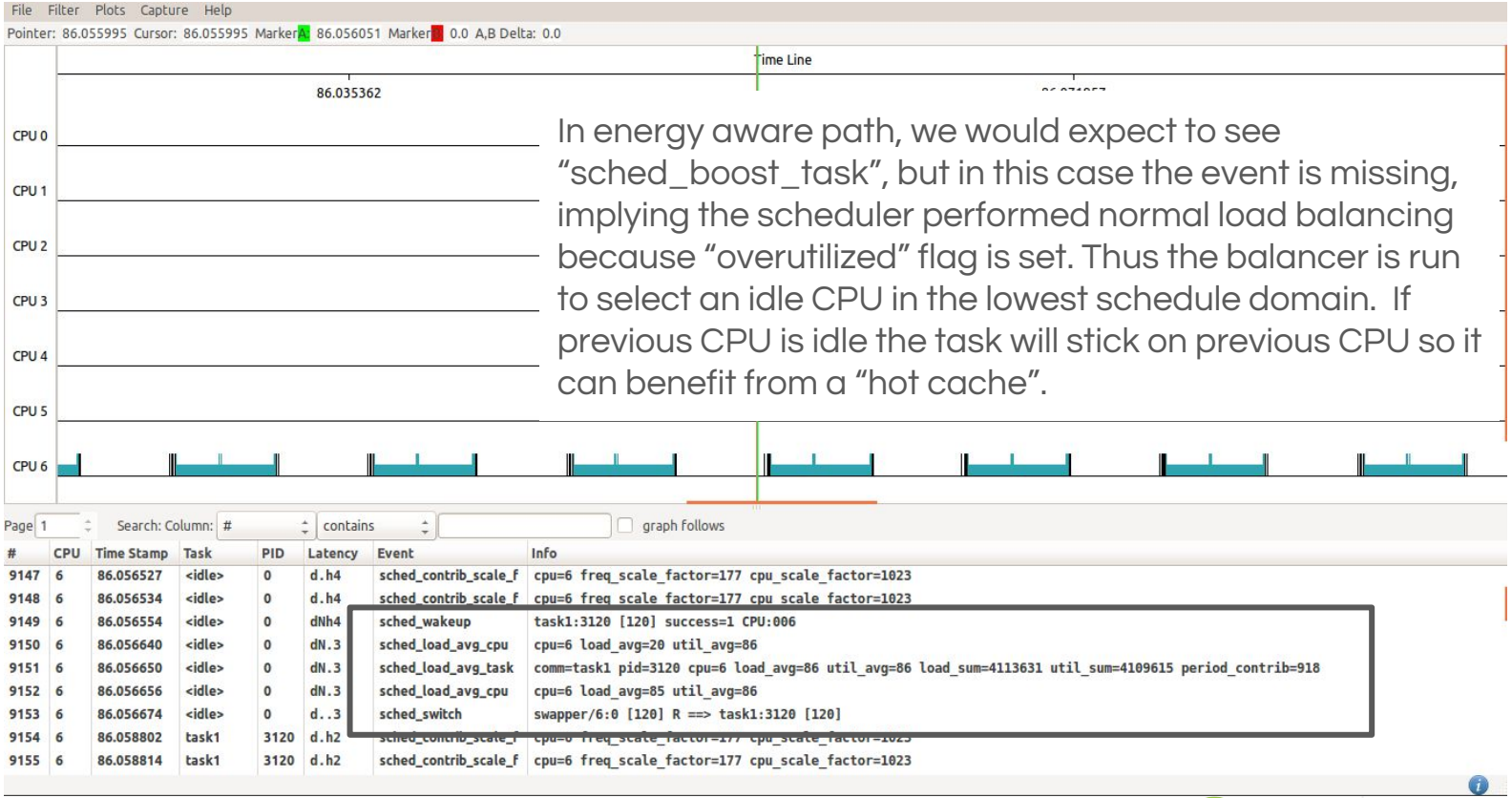
Analyze task's utilization signal

Filter only related tasks

PELT Signals for task utilization



Use kernelshark to check wake up path



In energy aware path, we would expect to see “sched_boost_task”, but in this case the event is missing, implying the scheduler performed normal load balancing because “overutilized” flag is set. Thus the balancer is run to select an idle CPU in the lowest schedule domain. If previous CPU is idle the task will stick on previous CPU so it can benefit from a “hot cache”.

The “tipping point” has been set for long time

```
static inline void update_sg_lb_stats(struct lb_env *env,
                        struct sched_group *group, int load_idx,
                        int local_group, struct sg_lb_stats *sgs,
                        bool *overload, bool *overutilized)
{
        unsigned long load;
        int i, nr_running;

        memset(sgs, 0, sizeof(*sgs));

        for_each_cpu_and(i, sched_group_cpus(group), env->cpus) {

                [...]

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

                [...]
         }
}
```

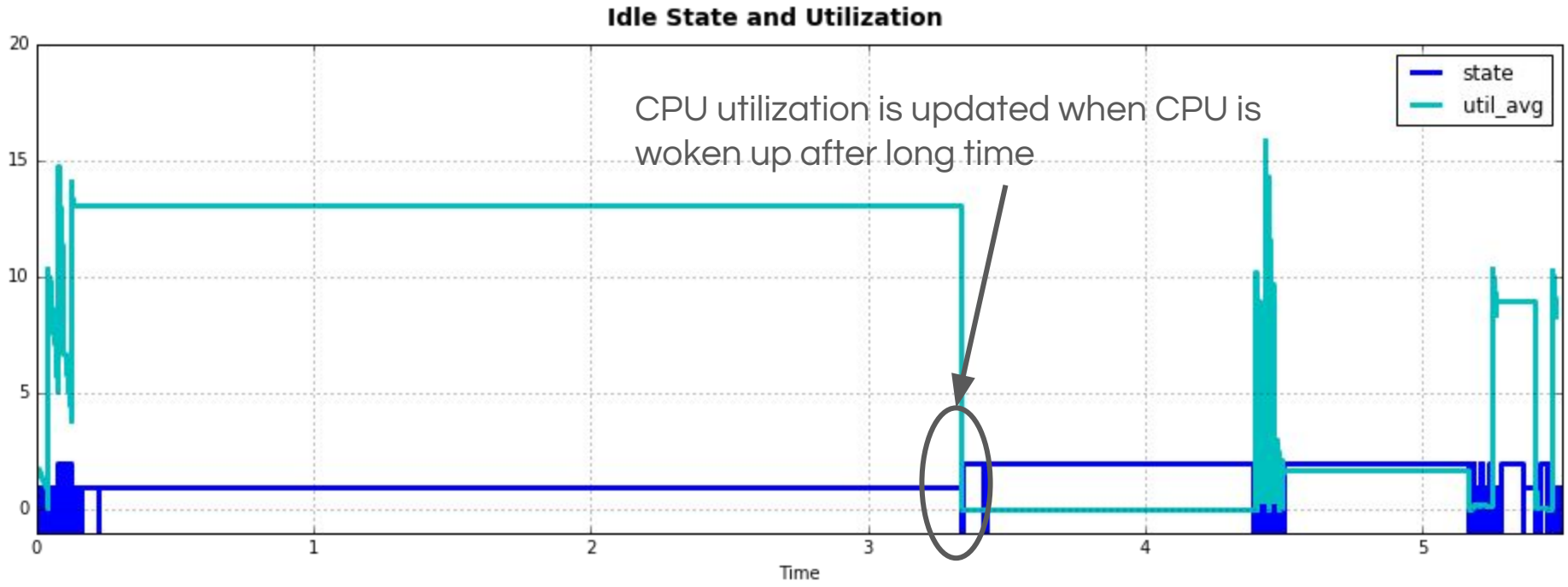
*overutilized is initialized as ‘false’ before we commence the update, so if any CPU is over-utilized, then this is enough to keep us over the tipping-point.

So need analyze the load of every CPU.

Plot for CPU utilization and idle state

```
def analysis_cpu_idle_state_and_util_avg(cpu):
    colors=['c', 'b', 'r']

    gs = gridspec.GridSpec(1, 1)
    fig = plt.figure(figsize=(16, 5))
    fig.subplots_adjust(top=0.92)
    plt.suptitle("Idle State and Utilization", fontsize=14, fontweight='bold')

    axes = plt.subplot(gs[0, 0])

    cpu_state = df_idle_state[df_idle_state.cpu_id == cpu][['state']]
    cpu_state[cpu_state.state == 4294967295] = -1
    cpu_state.plot(ax=axes, drawstyle='steps-post', color=colors[1], linewidth=3, ylim=(-1,20))

    load_avg_cpu = df_load_avg_cpu[df_load_avg_cpu.cpu == cpu][['util_avg']]
    load_avg_cpu['util_avg'] = load_avg_cpu['util_avg'] / 100
    print load_avg_cpu
    load_avg_cpu.plot(ax=axes, drawstyle='steps-post', color=colors[0], linewidth=3)

    axes.grid(True)
```

CPU utilization does not update during idle

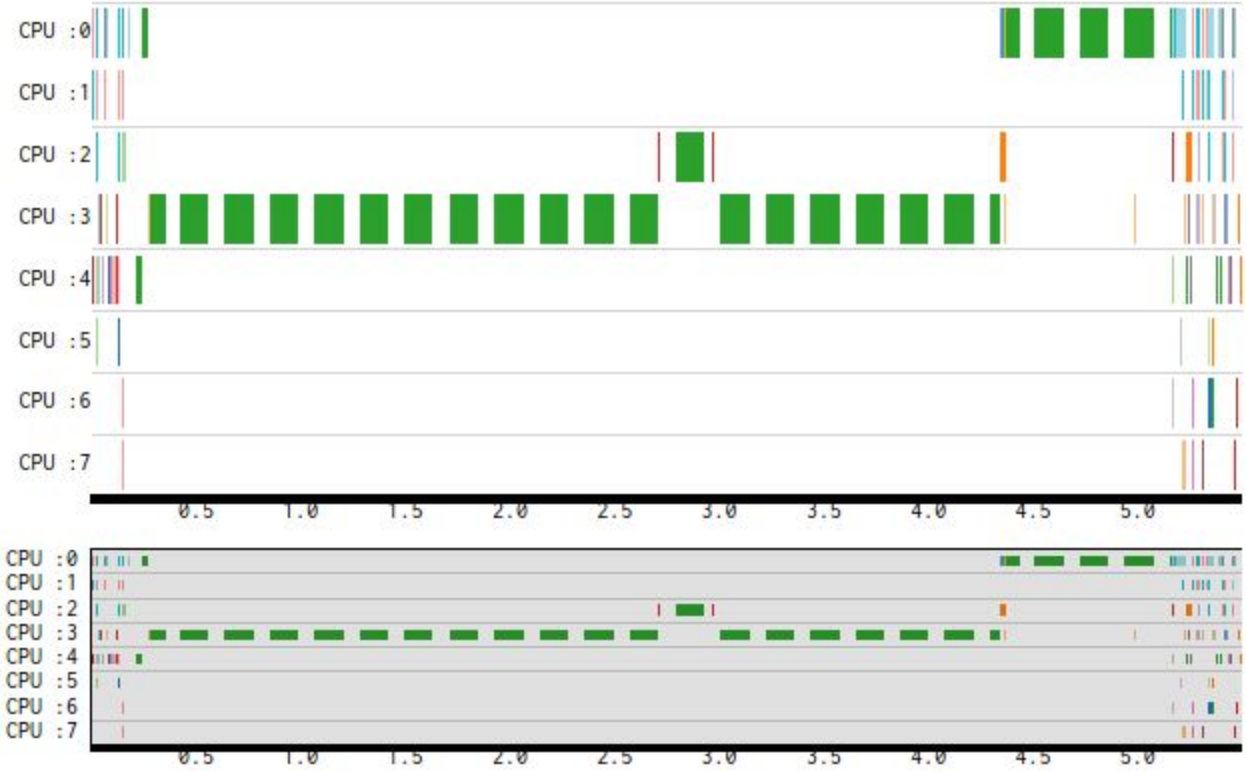


Fix Method: ignore overutilized state for idle CPUs

```
static inline void update_sg_lb_stats(struct lb_env *env,
                        struct sched_group *group, int load_idx,
                        int local_group, struct sg_lb_stats *sgs,
                        bool *overload, bool *overutilized)
{
        unsigned long load;
        int i, nr_running;

        memset(sgs, 0, sizeof(*sgs));

        for_each_cpu_and(i, sched_group_cpus(group), env->cpus) {

                [...]

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

                [...]
         }
}
```

Code flow is altered so we only consider the overutilized state for non-idle CPUs

After applying patch to fix this...



ENGINEERS
AND DEVICES
WORKING
TOGETHER

Agenda

- Background
  - Review of typical workflow for GTS tuning
  - Introduce a workflow for EAS tuning
  - Quick introduction of the tools that support the new workflow
- Worked examples
  - Development platform for the worked examples
  - Task ping-pong issue
  - Small task staying on big core
- Further reading

Related materials

- Notebooks and related materials for both worked examples
  - https://fileserver.linaro.org/owncloud/index.php/s/5gpVpzN0FdxMmGl
  - ipython notebooks for workload generation and analysis
  - Trace data before and after fixing together with `platform.json`
- Patches are under discussion on eas-dev mailing list
  - sched/fair: support to spread task in lowest schedule domain
  - sched/fair: avoid small task to migrate to higher capacity CPU
  - sched/fair: filter task for energy aware path
  - sched/fair: consider over utilized only for CPU is not idle

Next steps

- **You** can debug the scheduler
  - Try to focus on decision making, not hacks
  - New decisions should be as generic as possible (ideally based on normalized units)
  - Sharing resulting patches for review is highly recommended
    - Perhaps fix can be improved or is already expressed differently by someone else
- Understanding tracepoint patches and the tooling from ARM
  - Basic python coding experience is needed to utilize LISA libraries
- Understanding SchedTune
  - SchedTune interferes with the task utilization levels for CPU selection and CPU utilization levels to bias CPU and OPP selection decisions
  - Evaluate energy-performance trade-off
  - Without tools, it's hard to define and debug SchedTune boost margin on a specific platform

Thank You

#LAS16

For further information: www.linaro.org or support@linaro.org

LAS16 keynotes and videos on: connect.linaro.org