About the Need to Power Instrument the Linux Kernel

Patrick Titiano,
System Power Management Expert,
BayLibre co-founder.
www.baylibre.com
Today’s Special

- Introduction
- Power Instrumentation:
  - Why?
  - What’s needed?
  - What’s available?
  - What’s missing?
- Conclusion & Next Steps
- Q&A
Introduction

- A major issue the Linux Community faces regarding power management is the lack of power data and instrumentation
  - Dev boards missing probe points
  - Power Measurement equipment expensive / not affordable for many developers,
  - Poor power data publicly available
- This situation is not expected to change in the future
  - Believed that it is only of interest of a handful of developers, where actually everyone is concerned!
- This is forcing ad hoc/custom techniques to be used over and over again.
- Even if not much can be done on the HW side, power instrumenting the Linux Kernel with standard tooling could definitively help.
Power Instrumentation: Why?
Power Instrumentation: Purposes (1)

- Holy grail #1: enable dynamic measurement (estimation) of the platform power consumption / battery life, without any power measurement circuitry
  - Any developer could debug power management on any board, with no need of a special (expensive) board
Detect power leaks by dynamically monitoring (tracking) devices power state (Active / Idle / Disabled)

- Unnecessary running clocks
- Unnecessary running devices
- Inadequate CPUFreq/CPUIIdle states
  - CPU cores running too fast, low-power C-States not entered
- Unnecessary powered-on regulators
- ...

Power Instrumentation: Purposes (2)
Power Instrumentation: Purposes (3)

- Capture system power trace, and post-process it to
  - Generate use-case power statistics,
  - Generate power charts
- Enable more efficient power debugging
- Enable power consumption regression tracking automation
  - Integrate Continuous Integration (CI) frameworks (KernelCI, PowerCI, fuego, ...)
Power Instrumentation: Purposes (4)

- Model nextgen platform power consumption
  - Applying power data of next SoC revision to an existing power trace

- (... We could even imagine comparing platforms to platforms ... 😊)
Power Instrumentation: Purposes (5)

- Holy Grail #2: closed-loop power management policies
  - Prediction may be improved by measuring the “real” impact of heuristics decisions on platform power consumption
    - E.g. EAS (Energy-Aware Scheduler) platform knowledge could be extended beyond CPU cores
  - Could open the door to self-learning policies / IA / deep learning
    - No more need to fine-tune policies by hand, just let the policies learn the platform!
Power Instrumentation:

*What’s needed?*
What’s needed? (1)

1. SW Power Probe points
   - Regulator / Clock / Power Domain / CPU Frequency / CPU Idle / device / GPIO / …
   - Timestamped
   - power transitions
What’s needed? (2)

2. Power consumption data

- How much power is consumed by a given device in a given power state
  - SoC internal peripherals (CPU, GPU, RAM, UART, I2C, SPI, GPIO, ...)
    - E.g. UART devices consumes 5uW (*) when suspended, 100uW (*) when active
  - Platform peripherals (LCD display, wireless devices, flash devices, sensors, ...)
    - E.g. eMMC device consumes 500uW (*) when suspended, 40mW (*) when active

* Empirical data, for illustrative purpose only
What’s needed? (3)

3. Power Analysis Tools
   - Power trace plotting
   - Power trace statistics post-processing
   - Generic / Cross-platform Tools
     - Vendors already have some custom tools of their own, e.g.
       - Qualcomm’s Snapdragon Profiler (requires Android)
       - Google’s Android Systrace (may require Android too 😊)
Power Instrumentation:
What’s available?
FTrace Power Events (1)

- Kernel Probe Points
  - FTrace standard power events
    - RuntimePM events (idle/resume/suspend),
    - Clock Management events (enable/disable/set_rate),
    - CPU power management events (cpuidle/cpufreq/hotplug),
    - Suspend/Resume events,
    - Regulator events (enable/disable/set_voltage),
    - GPIO events (direction/value).
  - FTrace custom events
    - Specific for a given platform
To trace power events with FTrace

- Enable CONFIG_FTRACE, CONFIG_DYNAMIC_FTRACE flags in kernel .config file
- Mount debugfs
  # mount -t debugfs nodev /sys/kernel/debug
- Enable relevant events
  # echo 1 > /sys/kernel/debug/tracing/events/power/enable
- Empty trace buffer
  # echo > /sys/kernel/debug/tracing/trace
- Enable tracing
  # echo 1 > /sys/kernel/debug/tracing/trace_on
- Trace file /sys/kernel/debug/tracing/trace generated with enabled power events

* Note that debugfs interface is used for educational purpose here, but “trace-cmd” binary tool can be used.
Example of collected power trace

<table>
<thead>
<tr>
<th>TASK-PID</th>
<th>CPU#</th>
<th>TIMESTAMP</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>irq/676-lsm33d8-2917</td>
<td>002</td>
<td>d.2</td>
<td>clock_disable: gcc_bsp1_uqp6_i2c_apps_clk state=0 cpu_id=2</td>
</tr>
<tr>
<td>&lt;idle&gt;=0</td>
<td>001</td>
<td>d.2</td>
<td>cpu_power_select: idx1 sleep_time:211893 latency:91 next_event:0</td>
</tr>
<tr>
<td>irq/676-lsm33d8-2917</td>
<td>002</td>
<td>d.3</td>
<td>clock_disable: bsp1_uqp6_i2c_apps_clk src state=0 cpu_id=2</td>
</tr>
<tr>
<td>irq/676-lsm33d8-2917</td>
<td>002</td>
<td>d.2</td>
<td>cpu_idle: state=1 cpu_id=1</td>
</tr>
<tr>
<td>irq/676-lsm33d8-2917</td>
<td>002</td>
<td>d.2</td>
<td>cpu_idle: cpu_id=1</td>
</tr>
<tr>
<td>irq/676-lsm33d8-2917</td>
<td>002</td>
<td>d.2</td>
<td>rpm_suspend: 757a000.12c flags-d cnt=0 dep=0 auto-1 p-0 irq-0 clid=0</td>
</tr>
<tr>
<td>irq/676-lsm33d8-2917</td>
<td>002</td>
<td>d.2</td>
<td>rpm_return_int: rpm_suspend=0x36c/0x44c:757a000.12c ret=0</td>
</tr>
<tr>
<td>irq/489-d3-l2c-147</td>
<td>001</td>
<td>d.2</td>
<td>cluster: perf id=0 sync=0 clid=0 conf=0 cur=0 id=1</td>
</tr>
<tr>
<td>irq/489-d3-l2c-147</td>
<td>001</td>
<td>d.2</td>
<td>cluster: perf id=0 sync=0 clid=0 conf=0 cur=0 id=1</td>
</tr>
<tr>
<td>irq/489-d3-l2c-147</td>
<td>001</td>
<td>d.2</td>
<td>rpm_resume: 75b6000.12c flags-d cnt=0 dep=0 auto-1 p-0 irq-0 clid=0</td>
</tr>
<tr>
<td>irq/489-d3-l2c-147</td>
<td>001</td>
<td>d.2</td>
<td>rpm_resume: 75b6000.12c flags-d cnt=0 dep=0 auto-1 p-0 irq-0 clid=0</td>
</tr>
<tr>
<td>ksoftirqd/0-3</td>
<td>000</td>
<td>d.3</td>
<td>cpufreq_interactive_cputime: cpu=0 load=27 new_task_pct=0</td>
</tr>
<tr>
<td>ksoftirqd/0-3</td>
<td>000</td>
<td>d.3</td>
<td>cpufreq_interactive_cputime: cpu=1 load=16 new_task_pct=0</td>
</tr>
<tr>
<td>ksoftirqd/0-3</td>
<td>000</td>
<td>d.3</td>
<td>cpufreq_interactive_cputime: cpu=2 load=4 new_task_pct=0</td>
</tr>
<tr>
<td>ksoftirqd/0-3</td>
<td>000</td>
<td>d.3</td>
<td>cpufreq_interactive_cputime: cpu=3 load=9 new_task_pct=0</td>
</tr>
<tr>
<td>kworker/u8:10-3032</td>
<td>000</td>
<td>...1</td>
<td>memlat_dev_meas: dev: soc:com_memlat-cpu0 id=0, inst=227896, mem=783, freq=36, ratio=291</td>
</tr>
<tr>
<td>kworker/u8:10-3032</td>
<td>000</td>
<td>...1</td>
<td>memlat_dev_meas: dev: soc:com_memlat-cpu1 id=1, inst=17928, mem=58, freq=0, ratio=358</td>
</tr>
<tr>
<td>kworker/u8:10-3032</td>
<td>000</td>
<td>...1</td>
<td>memlat_dev_meas: dev: soc:com_memlat-cpu2 id=2, inst=37131, mem=132, freq=10, ratio=282</td>
</tr>
<tr>
<td>kworker/u8:10-3032</td>
<td>000</td>
<td>...1</td>
<td>memlat_dev_meas: dev: soc:com_memlat-cpu3 id=3, inst=0, mem=0, freq=0, ratio=0</td>
</tr>
</tbody>
</table>
FTrace Power Events (4)

- References:
  - https://www.kernel.org/doc/Documentation/trace/ftrace.txt
  - https://www.kernel.org/doc/Documentation/trace/events-power.txt
  - http://elinux.org/Ftrace
Power Instrumentation: What’s missing?
Missing Power “Database” (1)

- Power consumed by all devices of the platform, in any power state
- Not much data published so far, whereas critical
  - Usually only battery lifetime for selected use-cases
- Multi-platform database
  - Mandatory, to enable generic/standard tools

Example (empirical data, for illustrative purpose only)

```
# cat [...]/ftpwrdec/configs/arm64/arm/juno.pdb
# This is a sample power database file, in a human-readable format.
# Device power data format: name (as listed in ftrace), active_pwr (uW) suspended_pwr (uW)
devA, 10000, 10
devB, 1230000, 20
# CPU power data format: cluster id (as listed in ftrace), cpu id (as listed in ftrace), [frequency (MHz), power (uW)] ...
0, 0, [600, 300000], [900, 800000], [1200, 1200000]
1, 0, [200, 100000], [300, 150000], [500, 200000]
```

- Note Android already manages similar power database
  - power profile, defined in platform/frameworks/base/core/res/res/xml/power_profile.xml
Device Tree could also be a candidate

- Device Tree #1 purpose **IS** to describe the platform to the kernel,
- **Generic / Stable / Multi-platform,**
  - Mandatory for new platforms, existing platforms progressively converted
- **« Just a single attribute »** to be added to device attributes

```
# cat arch/arm/boot/dts$ cat omap4-panda-common.dtsi
/ {
 [...]
 &uart2 {
 [...]
   active-power = <200>; /* [1] */
   suspended-power = <5>; /* [1] */
};
&hdmi {
 [...]
   active-power = <7000>; /* [1] */
   suspended-power = <30>; /* [1] */
};
 [...]
```

[1] Empirical data, for illustrative purpose only
Power data in Device Tree could be reused by other Kernel components.
- FTrace
  - E.g. power data added to the trace log
- Kernel power management policies could reuse it
  - EAS (Energy-Aware Scheduler) / Closed-loop heuristics / deep learning algorithms

Also accessible from userspace
- /proc/device-tree/
- Existing libraries to read DT attributes, e.g. https://github.com/jviki/dtree

But
- Could be more difficult to maintain if part of the kernel
  - Longer review process
  - How would device tree maintainers test/validate the data?
FTrace « descrambling » tool (1)

- Static trace analysis
  1. Generate power statistics,
  2. Reformatted power trace for standard or dedicated plotting tools

- Multi-platform
  - To handle custom power events and reuse power consumption database

- Could be run directly on the platform or on a host machine

- Very useful for automation / Continuous Integration / power regression tracking
  - Build servers automatically run target use-cases, capture trace, generate the analysis, and generate reports highlighting regressions
  - Power consumption issues could be automatically detected upfront
FTrace « descrambling » tool (2)

Example

```bash
# ./ftpwrdec --plat=arm-juno mypowerftrace
Valid trace file found, descrambling it... done.
```

```
<table>
<thead>
<tr>
<th>Statistics</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Consumption</td>
<td>50mW</td>
<td>2000mW</td>
<td>530mW</td>
<td></td>
</tr>
<tr>
<td>CPU Loads</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU0</td>
<td>12%</td>
<td>42%</td>
<td>27%</td>
<td></td>
</tr>
<tr>
<td>CPU1</td>
<td>05%</td>
<td>35%</td>
<td>20%</td>
<td></td>
</tr>
<tr>
<td>CPU Idle Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU0</td>
<td>10ms</td>
<td>543ms</td>
<td>121ms</td>
<td></td>
</tr>
<tr>
<td>CPU1</td>
<td>44ms</td>
<td>876ms</td>
<td>465ms</td>
<td></td>
</tr>
<tr>
<td>CPU Frequencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU0</td>
<td>300MHz</td>
<td>800MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU1</td>
<td>300MHz</td>
<td>800MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Frequency Changes</td>
<td></td>
<td></td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>Active Devices</td>
<td>05</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Power Transitions</td>
<td>20</td>
<td>30</td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>Active Clocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock Transitions</td>
<td></td>
<td></td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>

[...]

'Mypowerftrace.xyz' data plotting file generated.
Done.
```

#
Static analysis of a trace is not sufficient

We need a visualization tool that could help us understand the dynamics of the system

- Like *kernelshark* does for cpu processes

Plotting in a smart way power events together with the power consumption

- Pointing a data point on the power consumption curve may highlight
  - Power consumption,
  - Current device power states,
  - Changes compared to previous data point,
  - ...
FTrace Power Visualization Tool (2)

* Empirical data, for illustration purpose only
FTrace Power Visualization Tool (2)

CPU1 Load (%)
CPU0 Load (%)
CPU Speed (MHz)
Active CPU Count
GPU Rate (MHz)
GPU State
Clock2 Rate (MHz)
Device2 State
Clock1 Rate (MHz)
Device1 State
Active Device Count
GPIO2 Value

Power: 1280mW
Active Devices: 10 (GPU, ...)
Active Clocks: 42 (GPU 600 MHz, ...)
Active CPU Count: 2 (1.6 GHz)
CPU Loads: 10% (0), 25% (1)
CPU Running Process:
Twitter (0), Pokemon Go (1)

Power Consumption (mW)

* Empirical data, for illustration purpose only
FTrace Power Visualization Tool (3)

- CPU1 Load (%)
- CPU0 Load (%)
- CPU Speed (MHz)
- Active CPU Count
- GPU Rate (MHz)
- GPU State
- Clock2 Rate (MHz)
- Device2 State
- Clock1 Rate (MHz)
- Device1 State
- Active Device Count
- GPIO2 Value

Power: 911mW
Active Devices: 9 (GPU, ...)
Active Clocks: (GPU 600 MHz), ...
Active CPU Count: 2 (1.0 GHZ)
CPU Loads: 10% (0), 0% (1)
CPU Running Process: Facebook (0), Idle (1)

Power Consumption (mW)

* Empirical data, for illustration purpose only
Summary

- **Bright side:**
  - Linux kernel has all infrastructure in place for power instrumentation
    - FTrace power / scheduling / performance / events
    - More relevant events may be relatively easy to be added
    - Tracing performance impact limited to RAM usage

- **Dark Side:**
  - Missing power consumption data
  - Missing standard analysis/plotting userspace tools
Next Steps

1. Collect more feedback and interest from experts during ELC,
2. Define the power database (incl. device tree vs userspace DB),
   - Probably the most difficult step as it will require a lot of experimentation, and support from vendors
3. Develop FTrace power events post-processing tool,
4. Develop power trace visualization tool, and...
   Make it the de-facto standard tool for power debugging 😉
Thank you!