Intel Corporation

Linux* Power Efficiency Analysis Methods

A look at power efficiency analysis methods under Linux environments

Christophe Zeitouny, Cagdas Akturan
7/25/2013

Contents

List of Figures

List of Tables

1 Introduction

The need for mobile devices with long lasting battery life has become more and more pronounced in an ever-connected world. Both high-tech societies and people living in rural areas with limited power resources can benefit greatly from more power-efficient computing devices. In this paper we provide a methodology for identifying the power efficiency issues and optimizing the power consumption for client platforms running on Linux*-based operating systems. The methods detailed here are applicable to the kernel, drivers, and user-mode software stacks.

Power efficiency has rapidly become a must-have feature for an operating system (OS) that is targeting mobile devices. For example, Windows 8*[8] [9] has improved greatly on its predecessor's power management capabilities, which allowed for Windows 8*-powered tablets and detachable laptops. Similarly, the Linux* kernel has also been adapted to support many power-efficiency features available in recent mobile platforms and devices. With the recent raging success of smartphones and tablets, Linux*-based operating systems are being optimized for these new battery-constricted platforms. Google's Android* offering [2] [5], Ubuntu*[10], now available more widely on smartphones and tablets, and the new Firefox* OS [1] are three such examples. Intel's recently released Ultrabook™ brand[4] is blurring the line between handhelds and laptops with many new power saving features that OS, driver, and application developers can utilize.

Solving power management issues and optimizing power efficiency is an iterative process. In this paper we start by outlining the high level methodology, followed by detailed power management analysis and optimization guidance for hardware, kernel, device driver and application stack along with practical examples.

2 Analysis Overview

Every optimization journey starts with the analysis of the current situation. Depending on the type of the target optimization goal, various types of up-front analyses need to be performed to assess the situation. For example, if the goal is to optimize the platform for longer gaming experience, several experiments need to be performed. However in addition to experiments representing the target scenarios, it is usually desirable to perform an idle analysis to make sure all power saving features on the platform work as expected and the platform has clean baseline power.

Note that, especially when performing idle analysis, the data collection process should be as non-intrusive as possible so as not to interfere with the platform's power management functionality. Next, we provide a few basic rules that should be followed when performing a complete system analysis.

2.1 Rules

2.1.1 Be conservative

If data collection is being performed on the target, especially for an idle scenario, frequent sampling can be very disruptive. Analysis programs should be instructed to update as infrequently as possible.

2.1.2 Be patient

Most power efficiency features do not take effect immediately after boot. Therefore, most power-efficiency problems require long-term monitoring. To perform such experiments, it is best to employ automation tools that can collect data over long periods. The use of such tools ensures that the experiments are repeatable, and that the results are reliable.

2.1.3 "General" is most often the sum of "local"

When performing a complete system analysis, there is usually no need to run all measurement tools at once as long as the steady state of the workload is understood.

2.1.4 A bottom-up approach works best

If the goal is to investigate both system and user application efficiency problems, it is best to start the analysis with a fresh-boot system where no extra services or user applications are started. Having a solid baseline helps greatly when trying to pinpoint potential problems without having to weed through many potential problems that can be due to user applications.

2.1.5 Proceed to active scenario slowly by allowing services and applications to run

If the platform idle baseline presents a clean power profile, the next step is to introduce startup services and user applications in a controlled fashion. The reason why a bottom-up approach works best is that it allows you to compare multiple measurements that have different services and applications running. Usually by comparing these measurements it is easy to pinpoint the source of the power efficiency issue, for example, a high number of GPU operations per second, a constantly decreasing amount of free memory, etc.

3 Methodology

Determining the root causes of power management problems requires an iterative analysis of the various system components. First, an understanding of the overall system behavior is needed. After that, focused tests on the problem areas need to be done to gather detailed data that would uncover the problem. During this process many software tools need to be employed in various stages. In the below diagram we present the recommended methodology to follow in order to systematically make progress towards determining the root causes of power management problems. In the following sections we will demonstrate this methodology with a practical example and provide information on the software tools that can be employed in various stages of analysis.

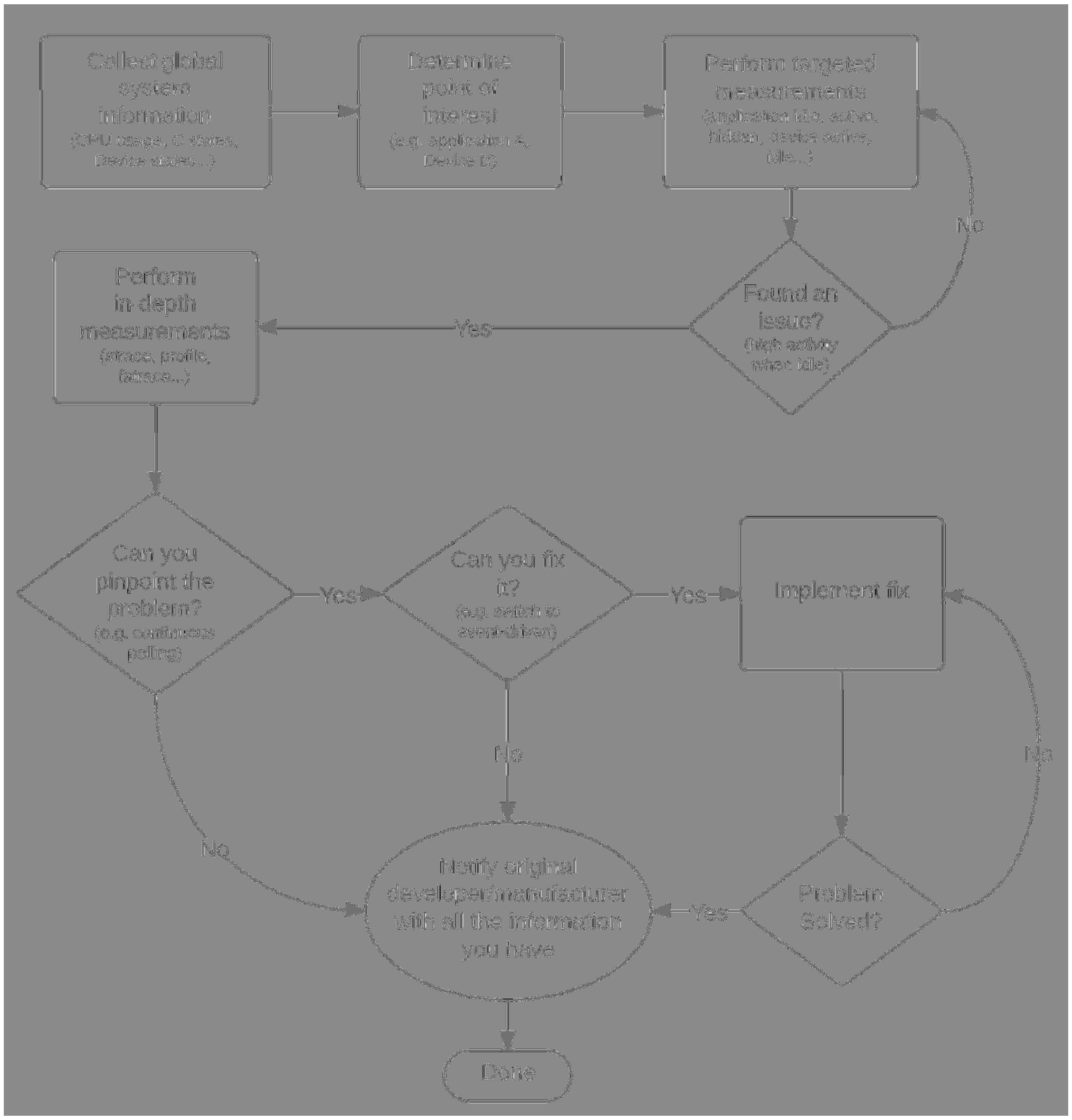

Collect global system information (CPU usage, C states, Device states...)
Determine point of interest (e.g. application A, Device B)
Perform targeted measurements (application idle, active, hidden, device active, idle...)
No
Found an issue? (high activity when idle)
Yes
Perform in-depth measurements (strace, profile, fatrace...)
Can you pinpoint the problem? (e.g. continuous polling)
Yes
Can you fix it? (e.g. switch to event-driven)
Yes
Implement fix
No
No
No
Notify original developer/manufacturer with all the information you have
Yes
Problem Solved?
Done

Figure 1: Investigative Process

3.1 Important Statistics

What follows is a list of system statistics that are essential to gathering a complete snapshot of the system's state:

- Processor utilization percentage by process
- Processor core and package C-states
- Processor P-states
- Processor wakeups per second
- Processor wakeups per process per second
- Interrupts per second for each registered IRQ
- Memory usage statistics by process
- Page faults per second
- Context switches per second
- Context switches per process per second
- Device power states
- Device link states
- GPU operations per second
- Network activity
- Disk activity

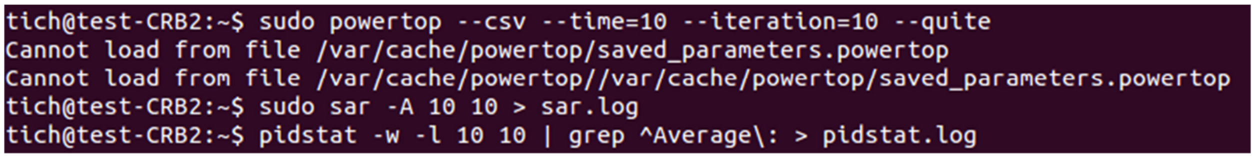
Unfortunately, it is often not possible to collect all the desired metrics using a single tool, and learning the intricacies of many tools may be time consuming. **Appendix A: List of Analysis Tools** provides a short summary of the Linux tools available today.

3.2 Analysis Example

To demonstrate how the metrics specified in the previous section can be used in a system analysis, the following analysis will go through the details of uncovering a power efficiency issue on a mobile platform.

For this example we use PowerTOP, SAR, and PIDStat to gather the needed statistics tools (refer to **Appendix A: List of Analysis Tools** for more information on these tools). PIDStat will only be used to get the number of context switches per process per second. Using these three tools, most of the metrics mentioned above can be gathered.

A simple example is shown below on how to collect these metrics:



```
tich@test-CRB2:~$ sudo powertop --csv --time=10 --iteration=10 --quite
Cannot load from file /var/cache/powertop/saved_parameters.powertop
Cannot load from file /var/cache/powertop//var/cache/powertop/saved_parameters.powertop
tich@test-CRB2:~$ sudo sar -A 10 10 > sar.log
tich@test-CRB2:~$ pidstat -w -l 10 10 | grep ^Average\: > pidstat.log
```

Figure 2: Command-Line Output

In the above example all three tools were instructed to gather statistics for 10 measurements with a duration of 10 seconds each. Assuming this is an idle analysis scenario, the first metric to check should be the measurement averages to ensure there is no excessive activity. If there is no issue with the average activity, we can start looking at the transient. While SAR and PIDStat automatically calculate these averages, you will have to perform these calculations manually for the 10 PowerTOP reports. The metrics provided hereafter are reformatted for clarity purposes. The actual format/nomenclature as reported by the tools might vary.

We start first with the general system statistics given by PowerTOP:

Total Wakeups/s	107.2
Total GFX Wakes/s	29.91
Total CPU %	1.93

Table 1: Initial Summary Statistics

While 107 wakeups per second isn't an excessive number, it is still elevated for a machine that is supposed to be idle. Note also the high graphics activity. This could be due to either the compositor (*compiz* in this case), or some application triggering the X server frequently. Next we look at the CPU usage breakdown, as reported by SAR:

All Processors	User	0.34 %
All Processors	Nice	0.00 %
All Processors	System	0.25 %
All Processors	I/O Wait	0.02 %
All Processors	Steal	0.00 %
All Processors	IRQ	0.00 %
All Processors	Soft IRQ	0.01 %
All Processors	Guest	0.00 %
All Processors	Idle	99.39 %

Table 2: Initial Processor Usage Statistics

The only thing to notice here is that the CPU is spending most of its time servicing user and kernel space processes. Since IRQs aren't taking up any significant portion of the CPU time, they are most likely just fine. In fact User, System, I/O Wait, IRQ, and Soft IRQ are the main values you should be inspecting. If any one of these is unreasonably high, then an issue most likely exists. In this case, however, no major issues are apparent, and we can proceed through the report.

IRQ 1	irq/sec	0.01
IRQ 57	irq/sec	0.8
IRQ 58	irq/sec	1.68

Table 3: Initial IRQ Statistics

The next metric to check are the statistics for the hardware interrupts.

Tip:

To determine which device each interrupt number corresponds to, you can read /proc/interrupts.

In this case, IRQ 57 corresponds to the AHCI controller (SATA disks), and IRQ 58 is the Ethernet interface.

While the disk activity might be a bit worrying, the constant light Ethernet activity is expected from a connected interface. To learn more about the disk activity, we should check the disk statistics.

Disk dev8-1	Transactions/s	0.89
Disk dev8-1	Read Sectors/s	0
Disk dev8-1	Write Sectors/s	23.36
Disk dev8-1	Avg. Req. Size	35.103
Disk dev8-1	Avg. Queue Length	0
Disk dev8-1	Avg. Wait (ms)	1.062
Disk dev8-1	Avg. Service Time (ms)	1.062
Disk dev8-1	Utilization (%)	0.08

Table 4: Disk Usage Statistics

From the table above, we can see that most of the disk activity is spent on writing. At this point we should suspect that this activity is caused by the statistics collection process and isn't actually something to worry about. However, to confirm this, the *fatrace* utility can be utilized to further investigate the source of the disk activity (see **Appendix A: List of Analysis Tools** for more information on *fatrace*). Notice how the number of transactions per second for a disk should match up to (or at least be a multiple of) the number of interrupts the corresponding AHCI controller is generating.

Tip:

To match up device IDs (of the form "devX:Y") with actual disk partitions, run the lsblk utility.

As a quick review, here's what we've gathered at this point:

- Total number of wakeups per second is slightly high, but not a significant cause for alarm.
- CPU is spending most of its time servicing user and kernel space programs.
- IRQ activity is normal. No device issues so far.
- Disk activity is most likely due to the actual statistics collection.

Moving on, we look at the processor C-State residency. In this case, we want the processor to be in C0 the least amount of time possible.

Package 0	C-State – POLL	0.00 %
Package 0	C-State - C1-HSW	0.02 %
Package 0	C-State - C3-HSW	0.20 %
Package 0	C-State - C6-HSW	0.08 %
Package 0	C-State - C7s-HSW	99.13 %

Table 5: Initial Processor C-State Statistics

Under optimal conditions, the processor should be spending more than 99% of its time in its deepest sleep state when the system is idle. This is the case in this example. However we should always try to maximize this residency, even if it looks "good enough."

To be able to theorize on the possible cause for this issue, we must take another look at what we know so far:

- Total wakeups per second is slightly elevated
    - Processor is either waking up too often or the offending task is performing long computations.
- Device activity is due to disk activity (IRQ 57) and idle network activity (IRQ 58)
    - Since the processor is not spending significant time in "I/O Wait" (see **Table 2**), it is most probably not the disk activity that is at fault here.

At this point, the list of suspects is (in descending order of plausibility):

- A user process that is waking up too often
- A user process that is performing extensive computations
- Very slow AHCI driver
- Kernel issue

To narrow down our search, we will have to look at the list of processes given by PowerTOP:

/usr/bin/gtk-window-decorator	Wakeups	3.86
/usr/lib/gnome-settings-daemon/gnome-settings-daemon	Wakeups	0.37
/usr/lib/unity/unity-panel-service	Wakeups	0.29
[4] block(softirq)	Wakeups	0.28
[7] sched(softirq)	Wakeups	0.4
[rcu_sched]	Wakeups	0.57
[usb-storage]	Wakeups	3.88
blk_delay_work	Wakeups	0.81
Clipit	Wakeups	64.57
Compiz	Wakeups	8.32
hrtimer_wakeup	Wakeups	2.17
menu_hrtimer_notify	Wakeups	2.94
od_dbs_timer	Wakeups	13.07
tick_sched_timer	Wakeups	3.22

Table 6: Initial Process Activity

Judging from the list above, the most apparent offender is *clipit*. Similarly, the PIDStat output in **Table 7** leads to the same conclusion, with the addition of the X server as a potential problem too. However, in most cases, the X server should be assumed to be as efficient as possible at idle (the same assumption goes for the kernel). Exceptions to these assumptions do arise, but are extremely rare. We then investigate the *clipit* process, which is a clipboard manager that displays an icon in the notification area. Given that we know the function that the process is supposed to accomplish, we can start coming up with theories as to what its effect on the platform could be. The thought process is as follows:

- It is a clipboard manager that is waking up 64.57 times per second
- The clipboard is usually maintained by the X server
- The application should then monitor the X server for any clipboard changes
- One (not very power efficient) way is to continuously poll the X server for the contents of the clipboard and compare them to the latest
- The X server is waking up (or being woken up) 30.9 times every second
- Given our assumption that the X server is most likely not the culprit, we can assume that it is the victim of the *clipit* application constantly requesting the clipboard contents
- The two major contributors to context switches are again *clipit* and the X server

UID	PID	cswch/s	nvcswch/s	Command
0	3	0.92	0.00	ksoftirqd/0
0	10	2.21	0.00	rcu_sched
0	13	1.30	0.00	ksoftirqd/1
0	18	0.48	0.00	ksoftirqd/2
0	23	0.90	0.00	ksoftirqd/3
0	38	2.33	0.00	kworker/1:1
0	58	2.54	0.00	kworker/0:1
0	83	1.27	0.00	kworker/3:1
0	261	5.87	0.01	usb-storage
0	273	0.12	0.00	kworker/0:1H
0	391	0.49	0.00	kworker/2:1H
0	1204	2.42	0.02	kworker/2:2
0	1392	30.90	0.83	/usr/bin/X :0 –core –auth /var/run/lightdm/root/:0 tcp vt7 -novtswitch
0	1403	0.25	0.00	/usr/lib/accountsservice/accounts-daemon
1000	1874	0.37	0.02	/usr/lib/gnome-settings-daemon/gnome-settings-daemon
1000	1910	25.10	1.33	compiz
1000	2006	3.98	0.11	/usr/bin/gtk-window-decorator
1000	2106	0.40	0.02	gnome-terminal
1000	2188	0.25	0.00	zeitgeist-datahub
1000	2215	36.09	0.91	clipit
1000	2224	0.25	0.00	/usr/lib/x86_64-linux-gnu/unity-lens-applications/unity-applications-daemon
0	2296	2.34	0.00	kworker/2:3
1000	2333	0.20	0.00	update-notifier
1000	2369	0.10	0.20	pidstat -w -l 10 10
1000	2370	2.88	0.00	grep --color=auto ^Average:

Table 7: Initial PIDStat Output

We can then safely say that while we're still not sure whether *clipit* is the sole contributor to the high number of wakeups, it is still not behaving in a power efficient way. To support that theory, we ran a new set of idle measurements with the *clipit* application not running.

UID	PID	cswch/s	nvcswch/s	Command
0	3	0.95	0.00	ksoftirqd/0
0	10	3.43	0.00	rcu_sched
0	13	2.01	0.00	ksoftirqd/1
0	18	0.66	0.00	ksoftirqd/2
0	23	1.17	0.00	ksoftirqd/3
0	38	1.73	0.00	kworker/1:1
0	58	2.19	0.00	kworker/0:1
0	83	1.39	0.00	kworker/3:1
0	261	5.84	0.00	usb-storage
0	273	0.18	0.00	kworker/0:1H
0	274	0.18	0.00	jbd2/sda1-8
0	391	0.49	0.00	kworker/2:1H
0	1204	2.26	0.00	kworker/2:2
0	1392	1.65	0.15	/usr/bin/X :0 -core -auth /var/run/lightdm/root/:0 tcp vt7 -novtswitch
0	1403	0.25	0.00	/usr/lib/accountsservice/accounts-daemon
1000	1874	0.57	0.06	/usr/lib/gnome-settings-daemon/gnome-settings-daemon
1000	1910	21.05	1.34	compiz
1000	2006	0.19	0.00	/usr/bin/gtk-window-decorator
1000	2106	0.59	0.00	gnome-terminal
1000	2188	0.25	0.00	zeitgeist-datahub
1000	2224	0.25	0.00	/usr/lib/x86_64-linux-gnu/unity-lens-applications/unity-applications-daemon
0	2296	2.12	0.00	kworker/2:3
1000	2333	0.20	0.00	update-notifier
1000	2394	0.10	0.26	pidstat -w -l 10 10
1000	2395	2.79	0.00	grep --color =auto ^Average:

Table 8: Final PIDStat Output

A quick glance over the PIDStat output (**Table 8**) reveals that the X server is no longer causing 30.9 context switches per second. It is down to a more manageable 1.65. Looking at the PowerTOP output, however, should give us a more comprehensive look at the machine:

Total	Wakeups/s	30.19
Total	GFX Wakes/s	0.08
Total	CPU %	1.17

Table 9: Final Summary Statistics

The total number of wakeups per second was cut down to 30, which is a good number for a machine running a full desktop environment. CPU utilization has also gone down from 1.93% to 1.17%. Similarly, the number of graphics-related wakeups was slashed to nearly 0, which is likely due to the fact that the X server is no longer being frequently polled.

IRQ 1	irq/sec	0.01
IRQ 57	irq/sec	0.62
IRQ 58	irq/sec	1.58

Table 10: Final IRQ Statistics

The status of the hardware interrupts is almost unchanged, which is understandable since the *clipit* application was not making use of any devices.

Package 0	C-State – POLL	0.00 %
Package 0	C-State - C1-HSW	0.01 %
Package 0	C-State - C3-HSW	0.05 %
Package 0	C-State - C6-HSW	0.00 %
Package 0	C-State - C7s-HSW	99.64 %

Table 11: Final Processor C-State Statistics

Looking at the C-State residencies for the processor, we notice an improvement in power efficiency. The CPU is now spending 99.64% of its time in C7.

/usr/bin/python /usr/lib/ubuntu-sso-client/ubuntu-sso-login	Wakeups	0.25
/usr/bin/python3.3 /usr/share/oneconf/oneconf-service	Wakeups	0.26
/usr/lib/gnome-settings-daemon/gnome-settings-daemon	Wakeups	0.37
/usr/lib/udisks2/udisksd --no-debug	Wakeups	0.1
/usr/lib/unity/unity-panel-service	Wakeups	0.14
/usr/lib/x86_64-linux-gnu/unity-lens-applications/unity-applications-daemon	Wakeups	0.26
[3] net_rx(softirq)	Wakeups	0.44
[4] block(softirq)	Wakeups	0.18
[7] sched(softirq)	Wakeups	0.81
[rcu_sched]	Wakeups	0.92
[usb-storage]	Wakeups	4.01
blk_delay_work	Wakeups	0.79
compiz	Wakeups	3.37
hrtimer_wakeup	Wakeups	1.01
menu_hrtimer_notify	Wakeups	1.16
od_dbs_timer	Wakeups	10.75
tick_sched_timer	Wakeups	2.82

Table 12: Final Process Activity

At this point, we know which application causes the system to misbehave. For information on how to analyze a specific application in hopes of optimizing its power efficiency, see the **Application-level Analysis** section.

Note that the process activity list (**Table 12**) now points us to other potential issues, like *od_dbs_timer* and *[usb-storage]*. Analyzing these issues is outside the scope of this example. This shows, however, that the optimization process is iterative. Once a major issue is resolved, minor issues start having more of a relative impact on the system.

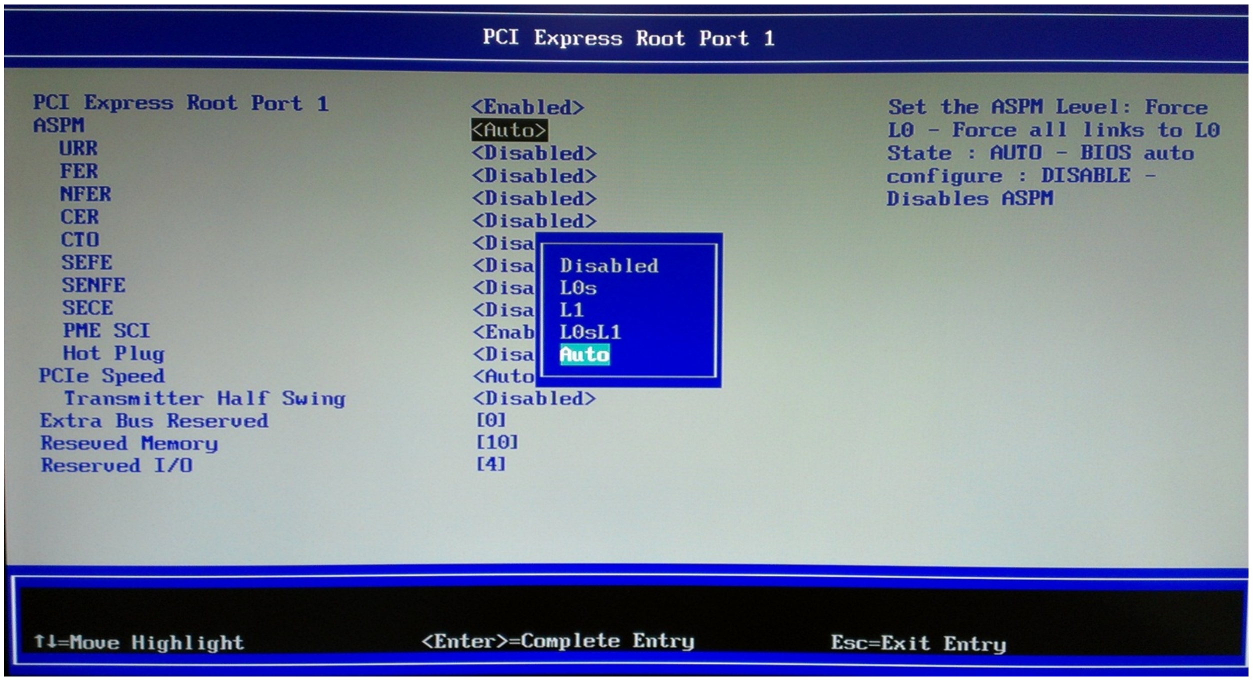
3.3 Focus Areas

While the practical example in the previous section presented an application-level issue, power efficiency problems can exist in any of the components on the platform's hardware and software stacks. This section breaks down these stacks and describes methods to analyze each.

3.3.1 Hardware Stack

The issues in the hardware stack are usually due to incompatibilities in the device configuration and driver software managing the device. In this section we will merely introduce how to perform a quick check on the hardware stack, as finding the root cause for hardware issues might be very complex and is beyond the scope of this paper.

To investigate hardware issues, an understanding of the device hierarchy (child devices connected) and device features is needed. Refer to manufacturer's datasheets to identify what configurations the device and its children support and how to configure them correctly for maximum power efficiency. Both *lspci* and PowerTOP can be used to gather information about device hierarchy and configuration. For example, the first screen shot below shows which low power link states are provided for a specific PCI Express* (PCIe) bus in the BIOS, while the second screen shot shows what states are supported by the device (using *lspci*). Notice that *lspci* reports that while the device supports both the L0s and L1 states, ASPM is disabled for this device. This means that either the OS or the driver disabled it.

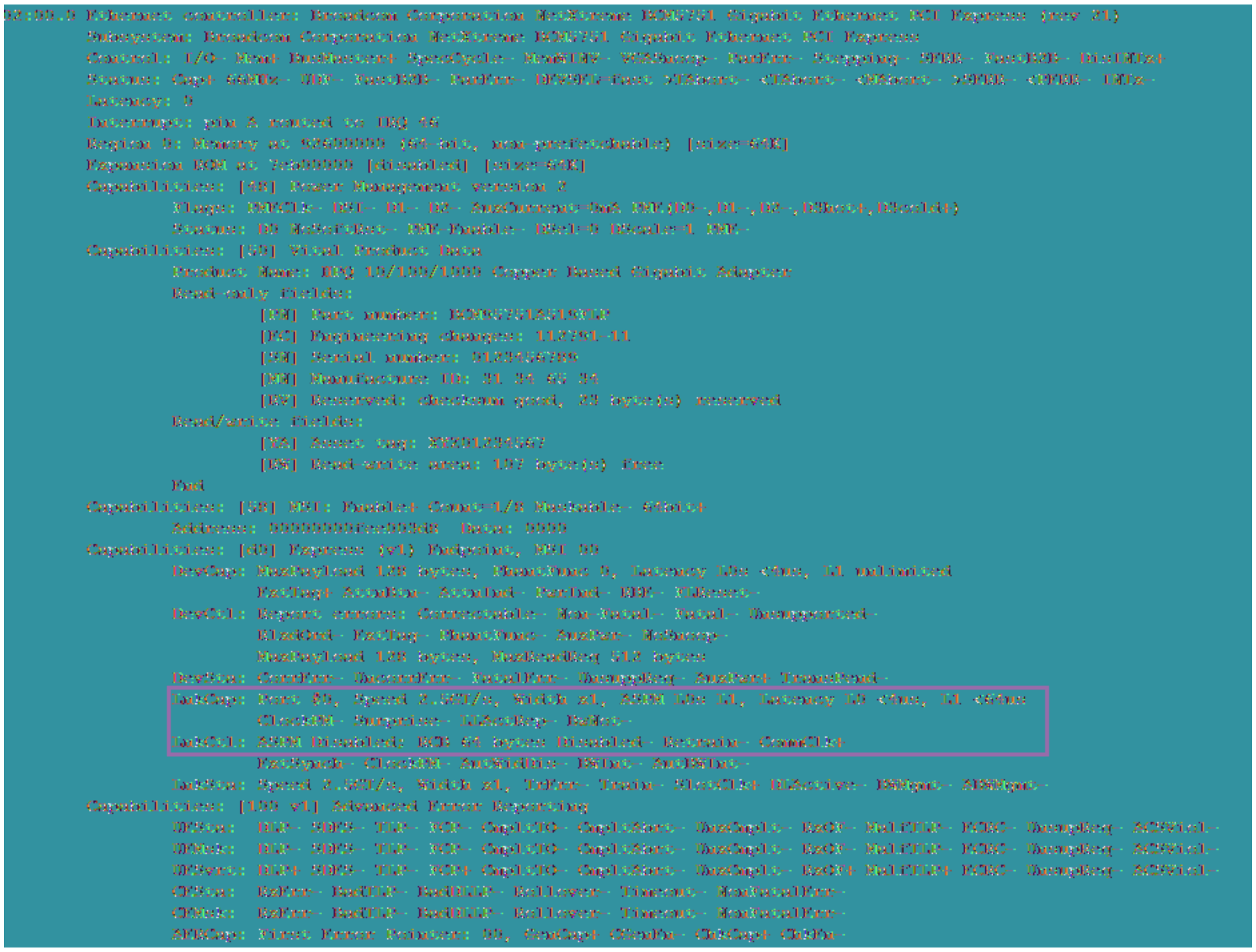


```
                              PCI Express Root Port 1

PCI Express Root Port 1          <Enabled>           Set the ASPM Level: Force
ASPM                             <Auto>              L0 - Force all links to L0
  URR                            <Disabled>          State : AUTO - BIOS auto
  FER                            <Disabled>          configure : DISABLE -
  NFER                           <Disabled>          Disables ASPM
  CER                            <Disabled>
  CTO                            <Disa
  SEFE                           <Disa  Disabled
  SENFE                          <Disa  L0s
  SECE                           <Disa  L1
  PME SCI                        <Enab  L0sL1
  Hot Plug                       <Disa  Auto
PCIe Speed                       <Auto
  Transmitter Half Swing         <Disabled>
Extra Bus Reserved               [0]
Reseved Memory                   [10]
Reserved I/O                     [4]

↑↓=Move Highlight         <Enter>=Complete Entry         Esc=Exit Entry
```

Figure 3: BIOS ASPM Settings



```
02:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5751 Gigabit Ethernet PCI Express (rev 21)
        Subsystem: Broadcom Corporation NetXtreme BCM5751 Gigabit Ethernet PCI Express
        Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR- FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0
        Interrupt: pin A routed to IRQ 46
        Region 0: Memory at 92600000 (64-bit, non-prefetchable) [size=64K]
        Expansion ROM at 7eb00000 [disabled] [size=64K]
        Capabilities: [48] Power Management version 2
                Flags: PMECLk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot+,D3cold+)
                Status: D0 NoSoftRst- PME-Enable- DSel=0 DScale=1 PME-
        Capabilities: [50] Vital Product Data
                Product Name: BRQ 10/100/1000 Copper Based Gigabit Adapter
                Read-only fields:
                        [PN] Part number: BCM95751A519FLP
                        [EC] Engineering changes: 112791-11
                        [SN] Serial number: 0123456789
                        [MN] Manufacture ID: 31 34 65 34
                        [RV] Reserved: checksum good, 23 byte(s) reserved
                Read/write fields:
                        [YA] Asset tag: XYZ01234567
                        [RW] Read-write area: 107 byte(s) free
                End
        Capabilities: [58] MSI: Enable+ Count=1/8 Maskable- 64bit+
                Address: 00000000fee003d8  Data: 0000
        Capabilities: [d0] Express (v1) Endpoint, MSI 00
                DevCap: MaxPayload 128 bytes, PhantFunc 0, Latency L0s <4us, L1 unlimited
                        ExtTag+ AttnBtn- AttnInd- PwrInd- RBE- FLReset-
                DevCtl: Report errors: Correctable- Non-Fatal- Fatal- Unsupported-
                        RlxdOrd- ExtTag- PhantFunc- AuxPwr- NoSnoop-
                        MaxPayload 128 bytes, MaxReadReq 512 bytes
                DevSta: CorrErr- UncorrErr- FatalErr- UnsuppReq- AuxPwr+ TransPend-
                LnkCap: Port #0, Speed 2.5GT/s, Width x1, ASPM L0s L1, Latency L0 <4us, L1 <64us
                        ClockPM- Surprise- LLActRep- BwNot-
                LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- Retrain- CommClk+
                        ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
                LnkSta: Speed 2.5GT/s, Width x1, TrErr- Train- SlotClk+ DLActive- BWMgmt- ABWMgmt-
        Capabilities: [100 v1] Advanced Error Reporting
                UESta:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC- UnsupReq- ACSViol-
                UEMsk:  DLP- SDES- TLP- FCP- CmpltTO- CmpltAbrt- UnxCmplt- RxOF- MalfTLP- ECRC- UnsupReq- ACSViol-
                UESvrt: DLP+ SDES- TLP- FCP+ CmpltTO- CmpltAbrt- UnxCmplt- RxOF+ MalfTLP+ ECRC- UnsupReq- ACSViol-
                CESta:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr-
                CEMsk:  RxErr- BadTLP- BadDLLP- Rollover- Timeout- NonFatalErr-
                AERCap: First Error Pointer: 00, GenCap+ CGenEn- ChkCap+ ChkEn-
```

Figure 4: Device ASPM Settings

If the devices seem to be correctly configured, then the issue might be due to defective hardware or it might be on the driver stack. If the issue is due to defective hardware, one way to find the root cause is to compare the device against another identical device by attaching it to the same system and showing that it is behaving correctly. Another way would be to instrument the device connections to check if all the signals are triggered correctly. However, this type of troubleshooting is often not possible due to lack of resources. It is then best to move on to checking the driver stack (see **Driver Stack**) and making sure that the issue is not due to a faulty driver. If the device is configured properly and all the software stacks above it are functioning according to specifications, yet the issue persists, then try downloading new firmware and/or contacting the original device manufacturer.

If an issue is found in the hardware stack, the **Device-level Analysis** section offers information on how to proceed.

3.3.2 Driver Stack

An issue in the driver stack usually presents itself in two possible ways:

- Bad/non-existent device power management
- Slow device performance, which also leads to decreased battery life

Therefore, the first step in discovering a driver issue is to inspect the platform at idle and check for any device activity. If the device is not being used, then it should be at sleep and/or reporting the longest supported latency. This could be determined by launching PowerTOP. If the device is going to sleep properly, then the driver is doing its job at power management. If not, see the **Driver-level Analysis** section for instructions on how to debug this.

Regardless of the outcome of the previous measurements, we still need to determine if the driver is inefficient. This can be done easily if a benchmark for the device already exists. Running the benchmark and comparing the results against the theoretical limits of the device can provide a general idea whether the device is performing well enough. If the benchmark falls short of the theoretical limit of the device, the same benchmark should be run on the same machine, however under a different operating system. If the second operating system produces a better benchmark score, then it is safe to assume that the Linux driver for that device is not as efficient as it could be. See **Driver-level Analysis** for more information on how to debug such an issue.

3.3.3 Application Stack

See **Important Statistics**

for an example on how to diagnose application-level issues. There are, however, some other tricks you can try in order to identify application-level power efficiency problems.

For example, the *eventstat* utility could be used to analyze the kernel timers and determine which applications are scheduling which timers. These timers force the processor to wake up to service them. The fewer and farther between the timers are, the more power efficient the platform becomes.

One other tool that could be of use is *pidstat*, which collects many useful statistics about a running process. One of these is the number of voluntary context switches per second that the process is performing. A context switch occurs whenever the process performs a system call. Context switches are expensive and should be minimized, at least at idle.

3.3.4 Kernel Stack

Given how well-maintained the Linux kernel is, the large majority of issues you could encounter with it are due to either regressions or unimplemented functionality. The kernel should only be suspected after checking the hardware, driver, and application-level stacks without finding any possible cause for the observed behavior.

Proper testing of kernel issues is a tedious process that involves checking the behavior on older kernels until one is found to exhibit the proper expected behavior. The issue is then denoted as a regression. This process can be somewhat automated by using the Phoronix Test Suite[7].

If, however, a kernel that exhibits the proper behavior is not found, the issue is most likely an unimplemented feature.

In both cases, it is best to search for an existing bug report for the issue before notifying the kernel maintainers. Additionally, adding whatever collected information to an existing bug report might help the maintainers reproduce the bug on their end, which would speed up the bug fixing process.

3.3.4.1 CPU Frequency Governor

The processor frequency is (for the most part) controlled by what's known as a P-State driver, namely *cpufreq* (for recent kernels and Intel processors, see *intel_pstate* below). This driver accepts different policies, known as governors. As of this paper, the four most well-known governors are:

- Performance: Activates the highest frequency at all times
- Ondemand: Sets the frequency based on the active workload
- Conservative: Same as *ondemand*, but places more weight on the low operating frequencies
- Powersave: Sets the processor to the lowest available frequency at all times

Note however that both the *performance* and *powersave* governors are detrimental to the system's battery life. While the reason is obvious for the performance governor, it is slightly more complicated for *powersave*. The latter forces the processor to operate at a very low frequency, which means that workloads take longer to execute. This forces the processor to stay active for longer, therefore burning through more power than it would have, had it been allowed to jump to a higher frequency. The *ondemand* governor is therefore the most optimal one.

Recent Linux kernels (starting with kernel 3.9) will incorporate a new scaling driver called intel_pstate that offers greater performance and power efficiency than *cpufreq* on recent Intel platforms. This new driver offers two governors:

- Performance: Activates the highest frequency at all times
- Powersave (default): Sets the frequency based on the active workload. This is the recommended setting for this driver, and does not suffer from the same drawbacks as *powersave* on *cpufreq.*

Tip:

The following command sets the governor for a specific CPU core:

```
echo ondemand > /sys/devices/system/cpu/cpuX/cpufreq/scaling_governor
```

4 Driver-level Analysis

As mentioned earlier, the two most prominent driver-level issues are: broken power management and poor driver performance. Each issue is explained in detail below.

4.1 Run-time Power Management Issues

4.1.1 Device Power Management Enable

If a device does not seem to be going to sleep, the first thing that should be checked is whether the kernel is instructed to enable run-time power management for this device.

Information about the device's power management status is stored under the *power* directory in the device's sysfs entry. For a PCI device, this can be found under:

```
/sys/bus/pci/devices/XXXX:XX:XX.X/power
```

Where XXXX:XX:XX.X is to be replaced by the device's PCI domain, bridge, bus, device, and function numbers respectively. These can be determined by running the *lspci* command as root.

Under the *power* directory, the *control* interface determines whether or not run-time power management is enabled for that device. Reading that interface can produce two possible outputs:

- on: The device is set to be always on and run-time power management is disabled
- auto: Run-time power management is enabled for this device

Fortunately, this interface is also writable. Writing one of the two commands above will set the run-time power management status for the device. Therefore, if it was set to *on*, it should be set to *auto*, and the original measurements should be rerun to see if the device now successfully goes to sleep.

Another interface to check is *autosuspend_delay_ms*, located under the same *power* directory. This interface controls how long (in milliseconds) the device should be idle before it is suspended. Some devices do not implement this interface, in which case an error is generated when the interface is read. If, however, the interface can be read, you should make sure it is not returning a negative number. Writing to this interface will set the autosuspend delay.

Tip:

For more information about the different interfaces in the power *directory, see* https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-devices-power

Most Ethernet devices offer Wake-on-LAN functionality that allows the device to wake the machine upon receipt of a specially crafted packet. This functionality requires parts of the Ethernet device to remain active even when the device is in its deepest sleep state, which negatively affects the battery life of the system (even when the system is completely turned off). If this functionality is not needed, you can use the *ethtool* utility to disable it, like so:

```
ethtool -s ethX wol d
```

4.1.2 Driver Support for Run-time Power Management

The PCI Express specification[6] requires that all compliant devices support at least the D0 and D3 sleep states. Whether transition to these is supported at run time is not dictated by the specification. It would therefore prove helpful to first check the **Device-level Analysis** section to make sure the hardware supports this feature.

Tip:

To determine which driver is loaded for a specific piece of hardware, run `lspci -k` *as root*

The source for the numerous device drivers usually resides in the *drivers* directory in the kernel source tree. The official kernel source repository can be found at http://www.kernel.org. However, many websites offer the kernel code online (e.g., http://lxr.free-electrons.com). The advantage that these websites offer is that they allow you to easily find the references to a certain function, variable, or data structure within the code. This makes the cross-referencing process a breeze, which speeds up code inspection time.

In the following example, we'll be investigating a Broadcom Corporation NetXtreme BCM5751 Gigabit Ethernet PCI Express card. The current symptoms are:

- Reports that D3hot and D3cold are both supported
- Only goes into D3 when the interface is brought down using `ifconfig eth0 down`
- The driver in use is *tg3*

What we need to figure out from the driver source is whether or not the driver implements support for suspending/resuming the device, and whether this is allowed at run time.

For PCI devices, the first thing to do would be to look for a structure of type `pci_driver` (defined in `/include/linux/pci.h`). Every PCI device driver has to fill this structure with specific information at initialization time. One of the uses of this structure is to declare the driver's suspend and resume functions. Historically, pointers to these two functions would be stored under "suspend" and "resume," respectively.

```
struct pci_driver {

    struct list_head node;

    const char *name;

    const struct pci_device_id *id_table;

    int  (*probe) (struct pci_dev *dev, const struct pci_device_id *id);

    void (*remove) (struct pci_dev *dev);

    int  (*suspend) (struct pci_dev *dev, pm_message_t state);

    int  (*suspend_late) (struct pci_dev *dev, pm_message_t state);

    int  (*resume_early) (struct pci_dev *dev);

    int  (*resume) (struct pci_dev *dev);

    void (*shutdown) (struct pci_dev *dev);

    int (*sriov_configure) (struct pci_dev *dev, int num_vfs);

    const struct pci_error_handlers *err_handler;

    struct device_driver    driver;

    struct pci_dynids dynids;

};
```

Code 1: PCI/PCIe Driver Main Structure

However, these pointers are deprecated and are only meant for suspend/resume functions that are called when the whole system transitions to a sleep state. In order to support run-time power management, new PCI drivers have to set the *driver* field to point to a valid structure of type `device_driver` (defined in `/include/linux/device.h`). Within that structure, the *pm* field needs to point to a valid structure of type `dev_pm_ops` (defined in `/include/linux/pm.h`). This last structure holds callbacks for all possible device power management statuses. The four interesting ones are: suspend, resume, runtime_suspend, and runtime_resume. The *runtime* variants are called when run-time power management is required.

Now that we know which power management functions need to be implemented by the driver so it can successfully change its power status at run time, we have to look for them in the driver implementation. What follows is a comprehensive list of possible scenarios:

- The driver uses the deprecated *suspend* and *resume* fields in the `pci_driver` structure
    - No run-time power management
- The driver explicitly initializes a valid `dev_pm_ops` structure, but does not set the *runtime_suspend* and *runtime_resume* fields.
    - No run-time power management
- The driver initializes a valid `dev_pm_ops` structure using the SIMPLE_DEV_PM_OPS macro
    - No run-time power management (this is the case in our example)
- The driver initializes a valid `dev_pm_ops` structure using the UNIVERSAL_DEV_PM_OPS macro
    - Run-time power management callbacks are the same as those for static power management
- The driver explicitly initializes a valid `dev_pm_ops` structure and sets the *runtime_suspend* and *runtime_resume* fields.
    - Run-time power management callbacks are present and correctly set

Starting with the "easy case" where run-time power management is properly implemented in the driver, the next thing to look for would be conditional entry points for device suspend. This requires us to check for any conditions in the driver code, and whether these conditions are easily matched with actual usage scenarios that could be mitigated. For example, a storage controller might not go into suspend mode unless all the disks connected to it are in full suspend. At this point, it would be reasonable to figure out why these conditions are not being met.

In the case where the run-time power management routines are not implemented in the driver, two possibilities exist:

- The device/platform does not support run-time power management
- The driver author omitted implementing the routines

First, we must make sure that both the device and the platform support run-time power management. See **Device-level Analysis**.

After completing the steps in **Device-level Analysis**, if the device is deemed to support run-time suspend then the power management function can (and should) be implemented. You can either do that yourself, or notify the original developer/maintainer for the device driver.

To implement run-time power management, the original device datasheet will be needed, along with a sizable amount of knowledge, discipline, and patience. If the driver handles multiple devices, special checks will have to be added to make sure the driver doesn't try to suspend a device that does not support it.

Tip:

To learn more about implementing run-time power management in drivers, see https://www.kernel.org/doc/Documentation/power/runtime_pm.txt

4.2 Performance Issues

If the device driver is not as optimized as it could be, device performance might suffer drastically. This matters for power efficiency since a slower device will take more time to perform the requested action, therefore spending more time being active.

The process of diagnosing performance issues consists of running a device benchmark, collecting performance data, and diagnosing the bottlenecks.

The process starts by determining a valid and consistent benchmark for the specific device under test. Some useful (albeit somewhat antiquated) lists of benchmarks available on Linux are:

- http://ltp.sourceforge.net/tooltable.php
- http://lbs.sourceforge.net/
- http://www.acnc.com/content.php?id=14

Once a consistent device benchmark is found, the next step is to profile the system while running it. This can be done using the Perf tools, as presented in **Appendix A: List of Analysis Tools.** Profiling the system while running the benchmark provides information that pinpoints the most time-consuming function calls.

Tip:

It is easier to make sense of the data if the profiling is run on a kernel that was built with debugging symbols included.

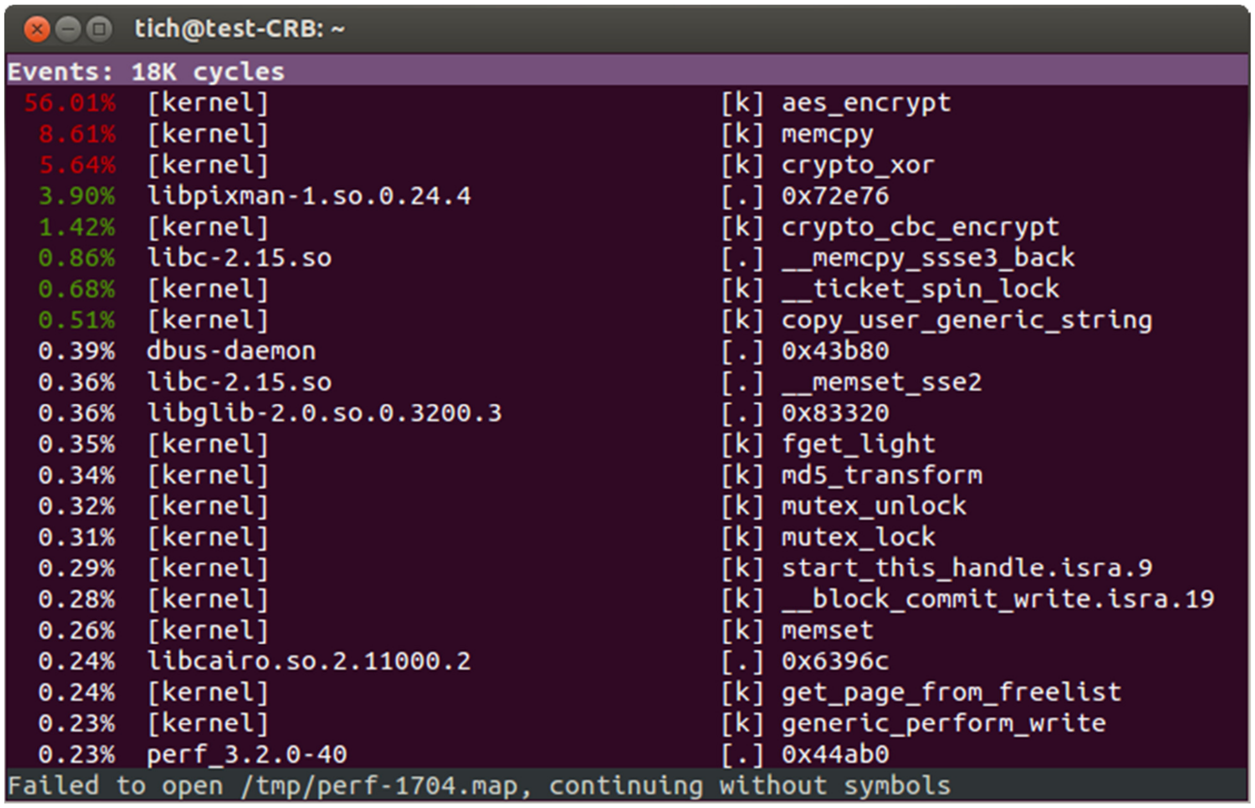
Consider the following example:

The following command was run on the system as a basic storage drive benchmark:

```
dd if=/dev/zero of=~/tempfile bs=1 count=500M
```

This command creates a file called *tempfile* and fills it with 500 MB of zeroes, one byte at a time. The file is written on an encrypted home partition.

While the original command was executing, we ran `perf top` in a separate terminal. The following is a screen shot of the output.

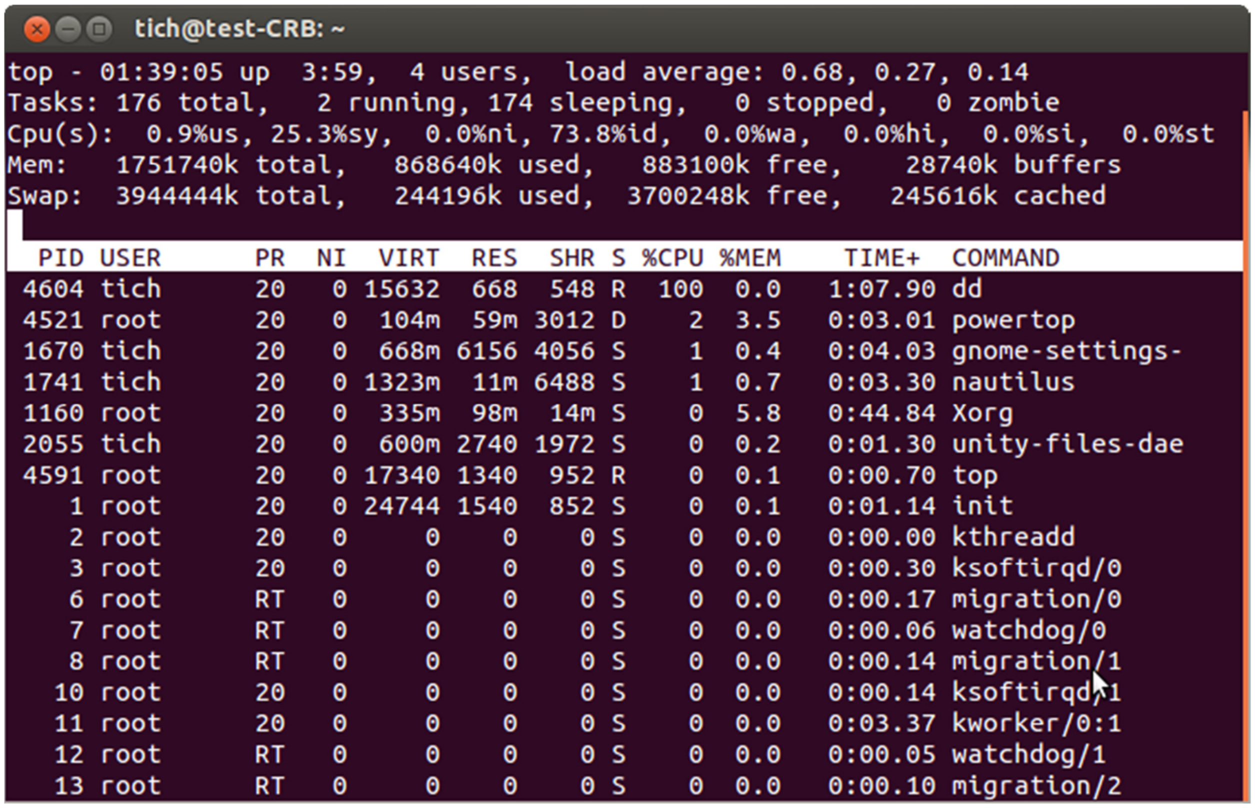


```
tich@test-CRB: ~
Events: 18K cycles
 56.01%  [kernel]                                   [k] aes_encrypt
  8.61%  [kernel]                                   [k] memcpy
  5.64%  [kernel]                                   [k] crypto_xor
  3.90%  libpixman-1.so.0.24.4                      [.] 0x72e76
  1.42%  [kernel]                                   [k] crypto_cbc_encrypt
  0.86%  libc-2.15.so                               [.] __memcpy_ssse3_back
  0.68%  [kernel]                                   [k] __ticket_spin_lock
  0.51%  [kernel]                                   [k] copy_user_generic_string
  0.39%  dbus-daemon                                [.] 0x43b80
  0.36%  libc-2.15.so                               [.] __memset_sse2
  0.36%  libglib-2.0.so.0.3200.3                    [.] 0x83320
  0.35%  [kernel]                                   [k] fget_light
  0.34%  [kernel]                                   [k] md5_transform
  0.32%  [kernel]                                   [k] mutex_unlock
  0.31%  [kernel]                                   [k] mutex_lock
  0.29%  [kernel]                                   [k] start_this_handle.isra.9
  0.28%  [kernel]                                   [k] __block_commit_write.isra.19
  0.26%  [kernel]                                   [k] memset
  0.24%  libcairo.so.2.11000.2                      [.] 0x6396c
  0.24%  [kernel]                                   [k] get_page_from_freelist
  0.23%  [kernel]                                   [k] generic_perform_write
  0.23%  perf_3.2.0-40                              [.] 0x44ab0
Failed to open /tmp/perf-1704.map, continuing without symbols
```

Figure 5: Profiler Output

Note that *aes_encrypt* is the most time-consuming function in terms of CPU time. However, we still need to make sure that the disk drive isn't the real bottleneck in this case. Since this is an I/O operation, we could check the I/O Wait time reported by SAR. This indicates how much time the CPU is wasting while waiting for I/O operations to complete.

In this case, we just ran "top" since it also provides the same information.



```
tich@test-CRB: ~
top - 01:39:05 up  3:59,  4 users,  load average: 0.68, 0.27, 0.14
Tasks: 176 total,   2 running, 174 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.9%us, 25.3%sy,  0.0%ni, 73.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1751740k total,   868640k used,   883100k free,    28740k buffers
Swap:  3944444k total,   244196k used,  3700248k free,   245616k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4604 tich      20   0 15632  668  548 R  100  0.0   1:07.90 dd
 4521 root      20   0  104m  59m 3012 D    2  3.5   0:03.01 powertop
 1670 tich      20   0  668m 6156 4056 S    1  0.4   0:04.03 gnome-settings-
 1741 tich      20   0 1323m  11m 6488 S    1  0.7   0:03.30 nautilus
 1160 root      20   0  335m  98m  14m S    0  5.8   0:44.84 Xorg
 2055 tich      20   0  600m 2740 1972 S    0  0.2   0:01.30 unity-files-dae
 4591 root      20   0 17340 1340  952 R    0  0.1   0:00.70 top
    1 root      20   0 24744 1540  852 S    0  0.1   0:01.14 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:00.30 ksoftirqd/0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.17 migration/0
    7 root      RT   0     0    0    0 S    0  0.0   0:00.06 watchdog/0
    8 root      RT   0     0    0    0 S    0  0.0   0:00.14 migration/1
   10 root      20   0     0    0    0 S    0  0.0   0:00.14 ksoftirqd/1
   11 root      20   0     0    0    0 S    0  0.0   0:03.37 kworker/0:1
   12 root      RT   0     0    0    0 S    0  0.0   0:00.05 watchdog/1
   13 root      RT   0     0    0    0 S    0  0.0   0:00.10 migration/2
```

Figure 6: top Output

There are many important things to note from this screen shot:

- "dd" is using 100% of one CPU thread on the system.
- The system is not spending any significant time waiting for I/O operations (indicated by *0.0%wa*).
- 25% of the time for the four CPU threads (which correlates to 100% on one thread) is spent in kernel space. This is corroborated by the information given in the first screen shot.

This allows us to ascertain that in this specific situation, the bottleneck is the disk encryption code in the kernel.

Once an issue is found with very high reproducibility, the intensive process of optimization can begin. Some recommendations to keep in mind while performing code optimization:

- Devise a series of tests that point out any issues with the modified code
- Find more benchmarks to run. Optimizing for one benchmark could degrade performance on a different one.
- Always test the modified code on a DIFFERENT machine (or a virtual machine). In this case, breaking the hard drive encryption/decryption code could render the home partition inaccessible.

5 Device-level Analysis

In order for device power management to be supported on a device-level, a couple of critical conditions should be satisfied:

- The device itself should support run-time power management
- If the device generates wakeup events, its interrupt pin should be connected and configured

5.1 Device Power Management

5.1.1 PCI/PCIe Devices

All PCI Express devices are required by the specification[6] to support at least two power states: D0 and D3. Furthermore, support for the ASPM link state L0s is also mandatory. To advertise these features to the OS, PCIe devices are required to implement two structures in their configuration space: the Power Management structure and the PCI Express Capability structure.

The former advertises the power management features supported by the device. What follows is some of the information reported by this structure:

- Current D-state
- Support for PME under different states
- Current PME status
- Support for the optional D1 and D2 states

Whether the device supports PME or not is an important piece of information for investigating power management issues. The next section goes into more detail about this.

The PCI Express Capability structure, on the other hand, advertises, among other things, the supported ASPM link states and whether ASPM is enabled on this device. ASPM support, while not critical, is good to have on a device. By powering down the serial interface, it also helps reduce the PCI root port's power consumption.

The simplest way to see these structures is by running `lspci -vvvv` on the system. Each advertised structure is listed under a *Capabilities* node. The supported features are followed by a + sign, while unsupported ones have a - sign. For example: if PME(D0+,D1-,D2-,D3hot+,D3cold+) is listed under the Power Management structure, you can deduce that a PME is supported only when the device is in D0, D3hot, or D3cold. The PME-Enable flag indicates whether PME is enabled at the moment. This will be useful in the next section.

Similarly, the ASPM features are listed under the *LnkCap* section in the PCI Express Capability structure. The *LnkCtl* section lists the current link status. The enabled ASPM states can be changed in the system

BIOS, which would affect the behavior of the device. For this reason, the BIOS should be checked first to make sure none of the ASPM features are disabled.

5.1.2 USB Devices

Similar to PCIe devices, USB devices also support multiple power management states. These states span from U0 to U3, where U0 is an active state and U3 indicates that the device is in its deepest sleep state. The intermediary U1 and U2 states are only supported by USB 3.0 devices, and are automatically negotiated between the device and the host. Consequently, there is currently no reliable method for software to determine when a device has entered U1 or U2.

Enabling power management for USB devices follows the same process as for PCI devices, as they share the same *power* interface under sysfs (refer to the **Device Power Management Enable** section).

5.1.3 SATA Devices

The SATA specification describes a power management feature for SATA devices, referred to as ALPM (Advanced Link Power Management). This feature allows the attached disk to switch between five distinct power phases:

- Active: Device is active and consuming the nominal amount of power.
- Partial: Device is in partial sleep. Wakeup latency is low (<10 µs). Marginal amount of power saving.
- Slumber: Device is sleeping, but power is maintained. Higher wakeup latency (<10ms). More power saving.
- DevSleep: Device is completely turned off, but power is maintained. Marginally higher wakeup latency (<20ms). Considerably higher power savings.
- RTD3: Device is completely turned off. Very high wakeup latency, with zero power consumption.

Thankfully, the *hdparm* utility provides Linux users with a simple command to get/set the ALPM policy for a SATA disk.

Tip:

hdparm -C /dev/sdX reads the disk drive's current state

hdparm -B /dev/sdX reads the disk drive's current APM policy

hdparm -B <policy> /dev/sdX sets the disk drive's APM policy. A lower policy indicates more aggressive power management

5.2 Wake Sources

There are only two ways a device can be woken up from a sleep state:

- The device generates a PME (a power management interrupt) that signals the OS to wake it up
- The operating system explicitly wakes the device by sending it an #RST signal

Since most devices on a machine act as interface devices, they had to be given a way to signal the operating system when any change occurred to their interface.

- SD card readers generate a PME whenever a card is inserted/removed
- USB controllers generate a PME when a device is plugged/unplugged
- Ethernet controllers generate a PME whenever a packet is received, a cable is unplugged, etc.

One way to determine under which conditions the device generates an interrupt is to check the device datasheet.

The reason why we are discussing device interrupts is because an improperly configured interrupt will cause the device to not be able to wake up from a sleep state. It is therefore imperative to make sure the interrupt mechanism is working properly before attempting to force the device to sleep (using driver modifications).

There are two ways for a device to signal a wakeup interrupt to the CPU. It could use either the legacy GPE mechanism or the native PCIe PME signaling route.

Native PCIe PME signaling involves routing the WAKE# pin of every PCIe device into its root bridge. This means that in order to signal a wakeup event, the device asserts its WAKE# pin, which notifies the root bridge of a wakeup event. The root bridge then sends an in-band interrupt message (using INTx or MSI/MSI-X) to the system, notifying it of this event. Given that the root bridge is generating in-band interrupt messages, this method does not require any special ACPI configuration. It does, however, require proper routing of the WAKE# signals to the root bridge (through a multiplexer).

The legacy GPE mechanism is inherited from PCI/PCI-X devices, where each device's WAKE# pin is routed directly to one of the GPIO pins on the platform. This setup bypasses the root bridge and is difficult to implement practically due to special routing considerations in the electrical board layout. Signaling PME events using GPEs requires some ACPI configuration to notify the OS of the correlation between each GPE and its corresponding device.

The first step in determining whether the device is properly configured to generate PME events is to find out which mechanism the platform is using to route PME interrupts. One quick way to do that is to read the contents of `/proc/interrupts`. If any line contains PCIe PME, the operating system is using native PCIe PME signaling. If not, then it is using the legacy GPE events.

If the operating system is using native PME signaling, the device PMEs should be automatically routed, and should work without any configuration. If, however, wakeup events are not working, then the most likely problem is with the way the root bridge forwards the PME to the system. If the root bridge is in D0, the interrupt is usually forwarded using MSI/MSI-X, which should also work perfectly without any special configuration. For this reason, you should try forcing the root bridge into D0 mode (see **Device Power Management Enable** for instructions). If this does not solve the problem and other devices under the same root port work perfectly, it could be a hardware issue with the device, which is out of the scope of this document.

If the operating system is using the legacy GPE method, each PCI/PCIe device should have a corresponding configuration in the DSDT ACPI table. Since each device is connected to the system via a specific GPE pin, this relationship should be advertised to the operating system so that it can correctly identify the device that generated a specific interrupt. To do that, the DSDT table should implement the _PRW method for each PCI/PCIe port. This function should return the address of the GPE pin that this device is connected to. See **Code 2: Sample _PRW Implementation** for a sample _PRW implementation for an onboard Ethernet controller. The sample method notifies the system that this controller is connected to GPE 0x0D.

```
Device (GLAN)

{

        Name (_ADR, 0x00190000)

        Method (_PRW, 0, NotSerialized)

        {

                Return (GPRW (0x0D, 0x04))

        }

}
```

Code 2: Sample _PRW Implementation

Furthermore, multiple devices can share the same GPE pin. To determine which device generated that interrupt, the operating system calls the associated _Lxx method, where *xx* is the GPE pin number. For example, if multiple devices are connected to pin 0x0D, then whenever that pin is asserted, the operating system calls the _L0D ACPI function. This function should determine which device generated an interrupt and report it back to the system using the Notify() mechanism. This is more clearly illustrated in the following example:

```
Scope (_SB.PCI0) {

      Device (GLAN) {

            Name (_ADR, 0x00190000)

            Method (_PRW, 0, NotSerialized) {

                  Return (GPRW (0x0D, 0x04))

            }

      }

      Device (RP01) {

            Name (_ADR, 0x001D0000)

            Method (_PRW, 0, NotSerialized) {

                  Return (GPRW (0x0D, 0x04))

            }

      }

}

Scope (_GPE) {

      Method (_L0D, 0, NotSerialized) {

            If (LEqual (\_SB.PCI0.RP01.PMES, One)) {

                  Notify (\_SB.PCI0.RP01, 0x02)

            }

            If (LEqual (\_SB.PCI0.GLAN.PMES, One)) {

                  Notify (\_SB.PCI0.GLAN, 0x02)

            }

      }

}
```

Code 3: Sample Code for Sharing GPEs

The example above is only intended for illustrative purposes. It is not guaranteed to be functional, or even compilable. For a complete reference on the _PRW and _Lxx methods, see the ACPI specification document[3].

If the legacy GPE interrupt mechanism is not properly functional, it is best to decode and check the platform's ACPI tables. If any error is found in the DSDT table, it is possible to perform the proper modifications and recompile the tables.

Tip:

To extract the DSDT ACPI table on a running Linux system, you can either use the acpidump *utility or run the following command as root:*

```
# cat /sys/firmware/acpi/tables/DSDT > dsdt.dat
```

The DSDT table can then be decompiled and recompiled using the iasl *tool.*

6 Application-level Analysis

Once the analysis focus has been narrowed down to one specific application, some more in-depth measurements of its running processes need to be performed. There are a multitude of tools that can help with this process. First, however, we must distinguish between the different power efficiency problems an application could suffer from. In this section we will introduce two types of issues that are commonly seen during our investigations and provide pointers and examples on how to debug them.

6.1 Busy-Wait Problems

One of the most predominant software problems on an idle machine is a busy-wait bug. A busy-wait condition occurs when the software is repeatedly checking a certain condition until it is satisfied. This keeps the CPU awake and is extremely power-hungry.

One busy-wait example we have encountered involved a service that is supposed to turn off the touchpad when the user is typing on the keyboard. However, in the absence of a touchpad, the service was stuck in a busy-wait loop, waiting for a touchpad to be connected to the system.

Now that we know what we should be looking for, we can start investigating ways to look for it.

As busy-wait loops are by definition repeatedly executing the same code, we can make the following assumption: The code inside the loop may include a syscall, a library call, or only the software's own code. It follows then that to accurately determine whether the program is executing a busy-wait loop, it suffices to monitor the system and library calls it is making. In the rare cases where only internal calls are made, we can use a profiler to monitor the code execution.

To get a list of the system calls placed by a running process, we can use the *strace* tool, for example:

```
strace -i -p NNNN
```

In the above command, NNNN is the PID of the process we need to analyze. The -i parameter instructs *strace* to output the instruction pointer at which the system call was placed. By visually monitoring for repeating sequences of syscalls that start at the same instruction pointer, we can tell whether the process is executing the same code indefinitely. Armed with the instruction pointer, we are indeed able to locate where exactly in the assembly code the program is looping. See **Converting a memory-mapped instruction pointer to the actual pointer**, followed by **Disassembling with *objdump***. If the program being inspected was built with debugging symbols enabled, then the disassembly should be more informative and will allow us to more easily determine the cause of this behavior in the original source code.

Monitoring the library calls placed by a program follows the same process as monitoring the system calls, except that you should use the *ltrace* tool instead.

As for profiling, you can use the Perf tools available on Linux. For example, the following command should work perfectly for this purpose:

```
perf top --call-graph --pid NNNN
```

In the above example, NNNN is the process PID. The top-ranking function call, or one of its parents, is the guilty function of the busy-wait bug. This method benefits greatly from rebuilding the process with debugging symbols enabled.

6.2 Non-Optimized Running State

A non-optimized application can manifest itself by generally being nonresponsive. On a system analysis level, this could manifest itself in a few different ways: high CPU C0 residency, high number of context switches per second, high number of page faults per second, etc. In the following sections, we list the types of analysis that can be performed to pinpoint the issues.

6.3 IO Activity Analysis

Using *fatrace*, all file accesses made by the process can be monitored. This helps determine whether the program is accessing the disk too often (which, apart from consuming CPU cycles, may keep the drive from achieving more power-efficient sleep states). A sample call to *fatrace* would be:

```
fatrace --timestamp | grep \(NNNN\)
```

Where NNNN is the process PID. Judging from the timestamps and the expected program behavior, you can make some guesses on whether the program is behaving correctly or not. One popular bug in this case would be repeatedly opening and closing the same file instead of holding it open for as long as the program needs it. Another popular, “not-so-power-efficient” behavior is constantly flushing a file handle. This circumvents the operating system’s built-in opportunistic flushing mechanism that buffers multiple file accesses and executes them when it is most appropriate to do so.

6.4 Utilization Analysis

Another approach to analyzing the program’s efficiency is using the Perf tools (outlined in **Appendix A: List of Analysis Tools**). The functions with the highest residency should be optimized first. Keep in mind, however, that knowing that a program is spending 76% of its time in one function isn’t very helpful if you don’t know how much processor time the program is consuming. There will always be a function that is used more than the others. Even if the program only calls it once every two seconds, and the execution time is only 1 millisecond.

Therefore, when optimizing an application using the profiler, PowerTOP and SAR should be used regularly to determine if the application is still inefficient.

7 Conclusion

Increasing the battery life of mobile devices offers more than environmental rewards. It brings about a better user experience for the users by allowing them to be more and more mobile. As we showed in this paper with examples, to achieve maximum power efficiency, all of the system components should be working in perfect harmony. This suggests that serious effort is needed on all fronts from platform hardware to user applications running on the platform. On the hardware level, chip vendors are working hard on enabling new power-saving technologies. However, more often than not, these features require special support from the operating system. Even then, one power-hungry application may be able to break most of the platform's power conservation features.

The methodology we described in this paper simplifies the process of power optimization and turns it into a set of targeted measurements and decision steps. The measurements and tools described in the paper do not require purchasing additional hardware resources or software tools, making power efficiency analysis and optimization accessible to all stakeholders from driver developers for peripheral devices, to application developers, to helpful hobbyists. Therefore power management analysis and optimization should be a standard step in kernel, driver, and application development.

8 Glossary

ACPI	Advanced Configuration and Power Interface. See http://www.acpi.info/
ASPM	Active State Power Management is a feature of the PCIe bus that allows it to downclock its numerous serial lanes to conserve power. The default running state is L0, with L0s and L1 being power-saving states. Whenever a device is in D3, however, the link state shifts to either L2 or L3. A properly functioning ASPM mechanism allows for great power savings since it allows both the device and the root PCIe bridge to consume less power.
Context Switch	The execution of instructions on the processor has to run within a context. Among other things, a context holds a process's registers and stack entries. This allows the operating system to switch between two running processes without losing their state during the transition. Usually the kernel has its own context. This means that whenever a process calls on a kernel function, the scheduler has to switch to the kernel's context before running that function. This is the most common cause of "voluntary context switches." "Involuntary context switches," on the other hand, occur whenever the scheduler decides to switch out the running process in favor of another process. This is the basic mechanism that allows multiple applications to run simultaneously on a system. Context switches, while not prohibitively expensive when in low numbers, can slow down a machine considerably if they occur too often.
Device Power States	PCIe devices support three general power states: D0, D3 hot, and D3 cold. The intermediary D1 and D2 states are optional and, more often than not, are ignored by the hardware manufacturers. A device in the D0 state is fully powered on and active. A device in the D3 hot state is suspended, yet still powered using the PCIe bus. A device in the D3 cold state is also suspended; however, power is only supplied via the Vaux rail of the PCIe bus.
DSDT	Differentiated System Description Table is part of the ACPI configuration for a system. It includes a platform-specific configuration that defines how the operating system is to interact with the hardware.
GPE	General Purpose Event
IRQ	Interrupt ReQuest denotes an interrupt, whether hardware or software.
Page Fault	A page fault occurs whenever a running process requests data that is not immediately available in the closest memory space. Usually, a page fault is mentioned in reference to a missing page in the RAM memory space. The occurrence of a page fault forces the kernel to perform some changes in the TLB entries. It may also force it to fetch some data from disk, which is a slow operation.

PID	Process ID, a unique number assigned to each running process that allows it to be uniquely recognized and addressed.
PME	Power Management Event is an interrupt raised by a device that indicates to the system it should change its power state.
Processor C-States	Processor idle states ranging from C0 to Cn, where C0 denotes an active processor. The deeper the C-State, the more features are disabled on the processor, allowing it to achieve lower power levels.
Processor P-States	Processor frequency states ranging from P0 to Pn, where P0 denotes a processor running at its rated frequency. The deeper the P-State, the more the processor is downclocked to achieve lower power levels. Note however that P-states only apply when the processor is in C0, actively executing instructions.
Processor Wakeups/s	The number of times the processor is woken up from an idle state. The processor wakes up whenever hardware or software interrupts are generated. Hardware interrupts are generated by input devices, PCIe devices, etc. Software interrupts, on the other hand, are generated by software timers expiring.
SCI	System Control Interrupt indicates an interrupt raised by a hardware device that indicates a system-level change. Examples include PME and platform power events.
Syscall	Short for "System Call." Denotes a call to a kernel function from a user-space program. For example, calling `read()` on a file descriptor results in a `sys_read` syscall that forces a context switch to execute this function in the kernel.

9 Bibliography

[1] *Firefox OS - Mozilla | MDN*. (2013, May 19). Retrieved from https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS

[2] Google Corporation. (2013, January 26). *Android*. Retrieved from http://www.android.com

[3] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. (2011, December 6). *ACPI*. Retrieved from http://www.acpi.info/spec.htm

[4] Intel Corporation. (2013, May 21). *Ultrabook(TM) Inspired by Intel*. Retrieved from Intel Corporation Website: http://www.intel.com/content/www/us/en/sponsors-of-tomorrow/ultrabook.html

[5] Open Handset Alliance. (2012, February 2). *Android Overview | Open Handset Alliance*. Retrieved from Open Handset Alliance: http://www.openhandsetalliance.com/android_overview.html

[6] PCI-SIG. (2013). *PCI-SIG - PCI Express*. Retrieved from http://www.pcisig.com/specifications/pciexpress/

[7] Phoronix Media. (2013). *Phoronix Test Suite - Linux Testing & Benchmarking Platform, Automated Testing Framework, Open-Source Benchmarking*. Retrieved from http://www.phoronix-test-suite.com/

[8] Sinofsky, S. (2011, November 8). *Building a power-smart general-purpose Windows*. Retrieved from http://blogs.msdn.com/b/b8/archive/2011/11/08/building-a-power-smart-general-purpose-windows.aspx

[9] Sinofsky, S. (2012, February 7). *Improving power efficiency for applications*. Retrieved from http://blogs.msdn.com/b/b8/archive/2012/02/07/improving-power-efficiency-for-applications.aspx

[10] Ubuntu. (2013). *The world's most popular free OS | Ubuntu*. Retrieved from http://www.ubuntu.com

10 Appendix A: List of Analysis Tools

Tool	Usage
PowerTOP	Intel-developed open-source tool that gathers information about: - CPU usage of the running processes: usage, wakeups/sec, power estimate - Total GPU operations/sec - CPU idle-state residencies: C0 …. Cn - CPU frequency-state residencies: P0 … Pn - Device idle-state residencies Use PowerTOP to also determine the most power-hungry processes. Website: https://01.org/powertop/
SAR	SAR is an open-source tool that is part of the *sysstat* package. It is more comprehensive than PowerTOP in that it shows many more system statistics: - Individual interrupts - Memory - Swap - Page faults - Disk I/O - Network I/O - Context switches - Thermal - Fans - etc. SAR is designed to collect data in the background using helper programs that are run as *cron* jobs every X minutes. However, you can request fresh data at any time using the *sar* command. If frequent data is needed, make sure to only request the minimal set of needed data. The more information is requested, the more work SAR has to perform in order to collect the results and, subsequently, the more impact it has on the results. Website: http://sebastien.godard.pagesperso-orange.fr/
Eventstat	Lists all the currently scheduled kernel timers along with the processes that scheduled them
Fatrace	A utility that provides disk I/O information per process on a file-access level. Website: http://www.piware.de/2012/02/fatrace-report-system-wide-file-access-events/
PIDStat	A utility that was developed by the author of SAR as a means to provide process-specific statistics. These statistics range from processor/memory usage to context switches, stack size, etc. As such, the amount of data reported by this utility can very easily become cumbersome. You must either carefully pick the statistics to collect, or filter the results to a few suspected processes. Website: http://sebastien.godard.pagesperso-orange.fr/
STrace	A Linux tool that collects information about all the system calls (and signals) that a process is making. Tracing system calls is very helpful in determining, for example, whether the program is stuck in an active-idle state where it is polling for events.

LTrace	A tool that traces library calls made by a process. Its usage is almost identical to *strace*. One extra parameter that is helpful is "-C", which provides demangling (http://en.wikipedia.org/wiki/Name_mangling) for library calls.
Lsof	A tool that displays information about all open file descriptors on the system. Note that a file descriptor on Linux does not *necessarily* describe an actual file. A file descriptor can point to a file, a socket, or a pipe (among others). After running *strace*, you could be left wondering what those file descriptors the process is polling actually point to. Thankfully, this information is provided by *lsof*.
Perf Tools	Perf is a set of performance analysis tools for Linux. These tools are very versatile and allow you to monitor hardware (CPU/PMU) counters and software behavior. The three tools of interest in this case are: - Perf-top - Perf-record - Perf-report Perf-top is best used when you suspect that the issue lies on the software-side. Since Perf-top displays relative CPU usage statistics, it gives no information regarding how much the process is actually hogging the CPU. It is therefore recommended to first run a different analysis tool (e.g., PowerTOP) to determine if there are any problems on the platform. If the CPU usage is high at idle, perf-top will come in handy in investigating that usage. When analyzing a program, notice a function call that exhibits a high CPU usage in perf-top. You must then check LTrace output for this same program to determine whether this is due to the function being called at a very high frequency (which indicates a problem with the calling program), due to the function inherently requiring a lot of CPU time (which indicates either a problem with the function itself), or both. Website: https://perf.wiki.kernel.org/index.php/Main_Page

11 Appendix B: Analysis Tricks

11.1 Converting a memory-mapped instruction pointer to the actual pointer

When a program is executed, the Linux kernel loads it into a memory location and begins executing the instructions from the program's entry point. GDB, STrace, and LTrace tools all deal with memory-mapped instruction pointers since they are tracing processes that are already in memory. However, you may want to know where exactly in the program the instruction is located, which is helpful when trying to optimize the program's code. When Linux kernel loads the program into memory, it stores the memory location that it was loaded into inside a special file under `/proc/<pid>/mem`. This file stores all the information about the memory mapping of the process. To find out where the actual process is located, execute the following command:

```
root$> cat /proc/<pid>/mem | grep /path/to/process | grep x
```

This results in an output that resembles:

```
7f29c1c0f000-7f29c60e8000 r-xp 00000000 08:05 1839248 /path/to/process
```

The first two hexadecimal numbers are the actual memory range where the process resides.

All this means is that if converting a memory-mapped instruction pointer given by GDB or perf-top (e.g., 0x00007f29c21cae11) to an actual instruction pointer is needed, you should subtract 0x7f29c1c0f000 (the base of the memory range) from it.

See **Disassembling with *objdump*** for an example explaining how to use *objdump* to disassemble a process.

11.2 Disassembling with *objdump*

Objdump is one of the Linux tools that can be used to disassemble objects from object files. We will focus only on -D option that is used as follows:

```
root$> objdump -C -D /path/to/process > disassembled.log
```

The above command disassembles the program, demangles (http://en.wikipedia.org/wiki/Name_mangling) the function names, and stores the output in a file called *disassembled.log*. It is now easy to look for the needed instruction pointer inside the *disassembled.log* file using any text editor.

Notices