Device Drivers Should Not Do Power Management

Chao Xu, Felix Xiaozhu Lin, and Lin Zhong
Rice University, Houston, TX

Abstract

We argue that device drivers are not the best place to implement power management policies for components on a system-on-a-chip (SoC). We present empirical evidence that device driver developers are inadequately implementing power management and show the information needed for good power management policies is available outside the device drivers. We implement a software central agent to infer the needed information and accomplish power management without device driver support. We further show that simple hardware support can eliminate the overhead of our approach and extend it to support all SoC components.

1 Introduction

Modern mobile systems employ a powerful, heterogeneous system-on-chip (SoC) as their primary computational engine. The SoC not only hosts the multicore ARM processor (main CPU) that runs the high-level operating system, e.g., Linux and iOS, but also integrate numerous specialized computational resources, e.g., DSP, GPU, and video/audio codec, along with controllers for I/O devices such as I2C, SPI and UART. Like other systems, the operating system exposes the functions of these specialized resources and I/O to user-space software via device drivers.

Because not all components of an SoC are used all the time, the operating system must carefully put idle components into low-power states for energy conservation and wake them up properly for the next job, a practice known as *power management*. Today a significant portion of this responsibility is shared by device drivers. The device driver developers decide at which line of the driver code to enable or disable a device.

The central, perhaps controversial, argument of this paper is: *we should relieve device drivers from the responsibility of implementing power management for SoC components*. Rather, they should only provide implementations of power state transition but leave the decision of state transition to a central agent. We support this argument in two parts.

First, we show that device drivers are bad places to implement power management for the following reasons. (*i*) Device driver developers tend to do a poor job in implementing power management. Even in the official Linux release, many SoC components have to wait for many months to see power management implemented in their drivers, e.g., UART, SPI and USB controller [1–3]. For those that do have power management implemented, the policies tend to be suboptimal, missing opportunities for further energy saving, e.g., watchdog timer [4] and GPIO [5]. This is because driver developers often focus on functionality, leaving power management as an afterthought. (*ii*) The power state of components on a modern SoC have close dependencies between each other. A device driver with poor power management policy may cause other power-hungry components from entering a low-power state and thus ruin the power management efforts of their drivers.

Second, we show that information needed for good power management policies is available outside the device drivers. We observe that power management of a SoC component requires the following three pieces of information. (*i*) the quality of service (Qos) requirement, e.g., wakeup latency; (*ii*) power and latency information about its low-power states; (*iii*) if the component has pending tasks. We observe that (*i*) is provided by other parties, e.g., user-space software or other dependent drivers and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
APSys '14, June 25-26, 2014, Beijing, China
Copyright 2014 ACM 978-1-4503-3024-4/14/06 ...$15.00.

(*ii*) is data sheet information, available offline from the vendor or by profiling. The reason that today's power management is implemented by device drivers is because (*iii*) is conveniently available in device drivers.

Our key insight is that (*iii*) can be made available outside the driver. We show that by monitoring memory access, we are able to infer (*iii*) to properly power manage SoC components with a software central agent. While this software approach comes with limitations, it demonstrates the feasibility of power management without device driver support.

Most importantly, we show that with small hardware modification, the limitations of our software implementation can be eliminated. In particular, modern SoCs already come with a hardware global power manager that performs low-power listening for components and implements clock/power gating for power management. We show that this hardware manager already has part of information (*iii*). With the information held by this hardware manager and small modifications to SoC components, power management of SoC components can be efficiently realized without device driver support.

2 Background

2.1 Mobile SoC basics

A modern mobile SoC consists of tens of hardware modules from general-purpose cores to specialized IP blocks including DSP, GPU, video/audio codecs and I/O controllers. Only some modules are capable of initiating interconnect communication with other modules; these modules are called *master* modules while others are called *slave* modules. Note that a master module can also initiate communication with another master module. Computational units, such as CPU, GPU and DSP, are usually master modules. Non-computational units such as I/O controllers are usually slave.

Device drivers run on the CPU and control other hardware modules, by accessing their registers.

2.2 Hardware support for power management

We next sketch the hardware support for power management available on mobile SoCs. Although our descriptions stem from our understanding of the TI OMAP4, a popular mobile SoC with abundant public information [6], we observe similar hardware support on other SoCs [7–12].

A hierarchy of modules and domains

Hardware modules may share clock and power supplies. This effectively organizes all modules into a hierarchy of domains.

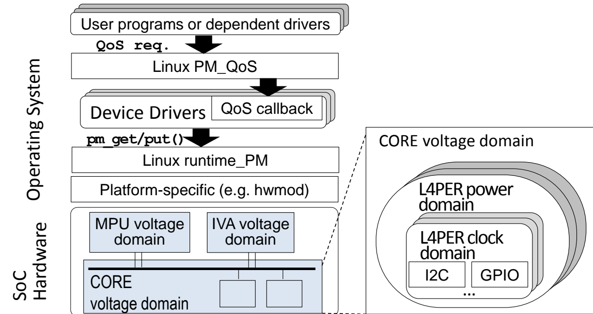

Figure 1: Resources (clock, power) are managed in a hierarchical way in SoCs. Users of a module make QoS requirements. Drivers adjust power management policy according to the requiements with the help from PM_QoS framework. Drivers carry out the policy with the help from runtime_PM framework.

Hardware modules: A module can be configured by software to be in either `enabled` or `disabled` mode. A module is functional only when it is `enabled`. When a module is `disabled`, it can be in one of several low-power states where clock and power can be gated to conserve energy. Note that software, e.g., device drivers, only determines the disabled/enabled mode; and hardware determines the specific low-power state for the disabled module.

Clock domain: a group of modules sharing the same clocks. Depending on whether the clocks are gated or not, the domain is either `active` or `inactive`. The state transition is controlled by hardware automatically.

Power domain: encompassing one or more clock domains, a power domain is a group of modules sharing the same power supply. The domain could either be `on`, `retention` (retention flip-flops are still powered on to preserve hardware state) or `off` (all flip-flops are powered off and hardware state is lost). The state transition is controlled by hardware. Software only controls which low-power state to enter.

Voltage domain: encompassing one or more power domains, a voltage domain is a group of modules that share the same voltage source. It can be either in `on`, `sleep` (supplying regular voltage, but limited current), `retention` (voltage drops to retention-level) or `off` (voltage drops to zero). The state transition is controlled by hardware. Software only controls which low-power state to enter.

The domains are the basic units of clocks management, power management, voltage management, correspondingly. The clock, power and voltage supplied to a module are only gated or lowered if the respective domain enters a low-power state under automatic hardware control.

Global power manager

A hardware unit called the global power manager centrally controls the clock/power/voltage supplies on the SoC. The global power manager is always-on because it is responsible to wake-up other modules or domains. When the entire SoC is suspended, it listens for wake-up events from outside and wakes up the SoC accordingly.

For hardware modules, the global power manager automatically gates the clock that drives the module's bus interface if there is no bus communication, and reactivates it if a new communication is initiated by a master module. However, only if the module is set to `disabled` by software can the global power manager gate all its clocks. For I/O controller modules that may serve as a slave in the I/O protocol (e.g., SPI), the global power manager detects events coming from outside of the SoC on behalf of the module if it is `disabled`. It then activates the module until the module generates an interrupt to CPU, so that the device driver has the chance to `enable` the module.

For clock domains, power domains and voltage domains, the global power manager automatically drives them to a low-power state if all the encompassed modules are configured as `disabled`; and it drives them back to functional if any encompassed module is configured as `enabled`.

2.3 Software support for power management

Most Linux systems implement power management in device drivers. Drivers are responsible to correctly set the mode of the modules, i.e., `enabled` or `disabled`. It is up to the hardware (global power manager) to automatically gate or lower the clock, power and voltage supplies to the module by driving the respective domain to low-power state. During initialization, the kernel sets the target low-power states of domains, as some of them have multiple low-power states.

Linux runtime PM framework

Linux provides the runtime PM framework as a unified interface to manage power state of modules on different platforms. *pm_runtime_get*() and *pm_runtime_put*() are the most important APIs. Calling them will increase and decrease a reference counter for a device, respectively. The framework maintains the reference counters for all devices. The framework calls platform-specific routines to set the module to `enabled` mode each time *pm_runtime_get*() function is called. But it only sets the module to `disabled` mode if the reference counter changes from one to zero. Another useful API is *pm_runtime_put_autosuspend*(), which allows the driver to specify a timeout after this API is called. It is useful to implement the timeout QoS requirement.

Linux PM_QoS framework

Linux provides the PM_QoS framework to allow users of a module to express QoS requirements, such as a timeout before setting the module to low-power mode, wakeup latency, never power off, etc. When a user changes the QoS requirement, the PM_QoS framework aggregates all users' requirements (e.g. maximum timeout) and calls the callbacks supplied by device drivers to update the PM parameters, such as the target low-power state, the timeout, etc. Note that clock gating and power gating are controlled outside of the device driver; from the driver's point of view, only one low-power state is supported: `disabled`. Thus, drivers are only able to fulfill simple QoS requirements, such as timeout. Currently the framework is not widely used for SoC modules.

Device drivers for SoC modules

Device drivers do power management with the help of the above two frameworks. Device drivers register callbacks with PM_QoS framework to receive notifications when users change QoS requirements. Ideally, before issuing a task on a module, the device driver calls *pm_runtime_get*() to set the module to `enabled` mode. After there is no pending task to run on a module, the device driver calls *pm_runtime_put*() to set the module to `disabled` mode. Failing to do so causes the module stuck in `enabled` mode. And it further causes the encompassing clock domain, power domain and voltage domain stuck in a high-power state.

3 Problems with device driver PM

Depending on device drivers to implement power management has the following two problems:

First, device drivers tend to do a poor job in implementing power management. For some drivers, power management comes as an afterthought. For example, drivers for the OMAP4's UART and USB EHCI controller, and those for the Samsung Exynos's keypad, SPI, and USB PHYs were not patched with power management implementations for several years after being introduced [1–3, 13, 14]. For drivers that do have power management implementations, the power management can be coarse-grained, missing further energy saving opportunities. For example, before two patches [4, 5], the OMAP4 GPIO driver and Watchdog driver enabled the modules in the driver *probe*() function and disabled them in the driver *remove*() function. Such implementations barely save any energy during runtime. The aforementioned power inefficiencies in drivers affect all mobile systems that use OMAP4 and Exynos.

Second, dependencies enlarge the damage caused by one driver that does poor power management. Because of

the hierarchical power management hardware support in SoCs, one poor device driver prevents the entire domain from entering low-power mode, ruining the power management efforts of the device drivers for other modules in the same domain. For example, because OMAP4 UART driver does not do power management, the entire L4_PER power domain is always on. According to our measurements, the L4_PER domain drains 17 mW more power when it is `on` compared to when it is in `retention`. There are also functionality dependencies between modules. For example, on OMAP4, DSP is kept on by its driver if the IVA is `enabled`.

4 Fundamental PM information

We study what information is essential to do good power management and whether power management can be done in places other than the device drivers.

A good power management policy saves as much energy as it can while it fulfills the QoS requirements specified by the users. Much research focuses focus on designing algorithms to make good power management policies, e.g., [15–17]. These algorithms require complex information like the workload statistics and therefore are not commonly used in practice. In Linux, the power management algorithms used by drivers are usually simple. They just translate users' requirements to PM parameters according to the data of different low-power states (currently SoC module drivers only support one low-power state, `disabled`), or set the timeout to the value given by PM_QoS framework, etc. The algorithms require the following three pieces of information:

(*i*) The QoS requirements from the users: The QoS requirements are the inputs to the power management algorithms. They affect when and to what power state a device is transited.

(*ii*) Data of the device in different power states: These are another part of inputs to the power management algorithms. The data include power consumption in all hardware states, the latency and energy consumption for transition between two states.

(*iii*) Whether a device has pending tasks: Only when a device has done *all* pending tasks, the power management policy sets the device to disabled, possibly after a timeout. When there is a new pending task for a disabled device, it needs to be enabled. Note that the functionality of a device is not damaged if it is set to disabled once it finishes *one* task and set to enabled before it starts next task. But this incurs unnecessary power state transition overhead. So a good power management policy keeps the module enabled until it finishes *all* pending tasks.

We observe that (*i*) is provided by other parties, e.g., user-space software or other dependent device drivers. (*ii*) is static information and is available offline (from the vendor or profiling). (*iii*) is conveniently available in device drivers and that is the reason that today power management is implemented by device drivers. (*iii*) is available in device drivers because they usually either maintain a queue of pending tasks or a counter of users that have tasks running on the device. Device drivers can know whether there are pending tasks by checking if the queue is empty or if the counter reaches zero.

Our key insight is that information (*iii*) can be made available in other ways. We next demonstrate that they can be inferred by monitoring memory access.

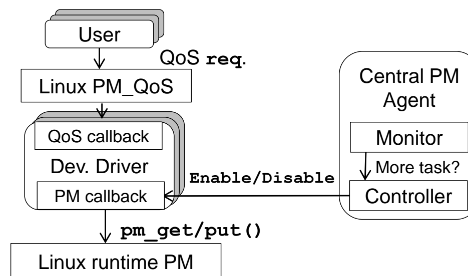

Figure 2: The central PM agent calls PM callbacks provided by the driver to enable/disable a module. PM QoS practice remains the same.

5 Central PM agent

To avoid mishandled power management in the drivers, we advocate the use of a central PM agent. In this section, we describe one way to realize the agent. We provide early results validating its effectiveness and discuss hardware support that can improve its performance.

5.1 Inference of pending tasks

To implement power management outside device drivers, one has to know if there is pending work for the target module. While one can ask device drivers to export this information, we find it can be inferred with reasonable accuracy and overhead, without device driver support.

Our critical observation is that (*i*) on ARM-based SoC, all modules are memory-mapped, i.e. the registers of the modules and memory are mapped to the same address space and (*ii*) typically, when a module has pending tasks, CPU frequently accesses its memory-mapped registers.

We use an example on the I2C controller to demonstrate our second observation. To prepare a read/write transaction, the device driver configures the I2C registers to setup the address of the message buffer, the length of the message, etc. During the transaction, the module frequently interrupts the CPU. The driver handles interrupts by reading/writing the IRQ status register. When the

transaction is completed, the CPU receives a last interrupt and then reads the IRQ status register. Register access occurs throughout the utilization cycle of the I2C module.

This suggests the module has no pending task if its registers have not been accessed for a certain time d. With a smaller d, the inferred moment is closer to the actual moment the module finishes all tasks. But to avoid false inference, d needs to be greater than the largest register access interval when a module is being used. Given the largest register access interval occurs when the module is using DMA to transfer data, d has to be greater than the longest DMA duration.

It is easy to detect when a non-functional module has a new task, which is indicated by the first register access after it has been set to `disabled` mode.

Note that due to the nature of our inference algorithm, the moment the algorithm infers a module has finished all tasks can be at most $2d$ time later than the actual moment, and thus losing some power saving opportunities. However, using a central PM agent prevents power bugs such as an entire power domain is kept on by a driver with poor PM. Hence, the tradeoff is worth paying.

5.2 Design of a central PM agent

The design of the central PM agent consists of two parts: the monitor part and the controller part. The monitor infers whether a module has pending tasks. The controller calls the PM callbacks provided by the driver to set the module to `enabled`/`disabled` mode based on the inference of the monitor.

Figure 2 shows the relationship between the central PM agent and existing Linux PM frameworks. For QoS support, a driver does not need to change its current practice, i.e., it registers callbacks with the PM_QoS framework to receive notifications about user requirement changes. The driver still calls runtime_PM APIs to implement power state transitions. The only modification to the driver is that it has to wrap these calls and provide them to the central PM agent as PM callbacks. For the central PM agent to manage a module, the module's device driver needs to register its PM callbacks with the central PM agent, without modification to the central PM agent itself. This gives driver developers the flexibility to decide if they want the central PM agent to manage their modules.

5.3 OMAP4-based implementation

Because modules are memory-mapped in ARM, we implement a Linux loadable kernel module to monitor the access to a module's registers on OMAP4 via memory exception. Figure 3 illustrates how the central PM agent works. During initialization, the monitor removes the read and write permissions of the memory pages mapped to the registers of the target module and marks all modules as "idle". It then sets up a timer to check whether the

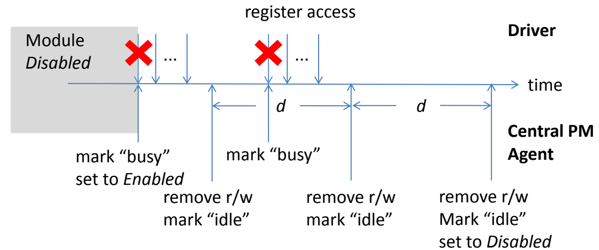

Figure 3: The central PM agent sets the module to `enabled` mode if the module was `disabled` when a memory exception occurs. It sets the module to `disabled` mode if the module has not been accessed for d.

module has pending tasks every d interval. Given the highest bandwidth of DMA on OMAP4 is around 100 MBps, we choose $d = 100$ ms, which allows 10 MB of data to be transferred per interval. Because the largest DMA transaction of SoC modules occurs when the imaging subsystem handles a photo, the size of which is still smaller than 10 MB, $d = 100$ ms is large enough.

When a memory exception occurs, the monitor first checks if the module was marked as "idle". If so, it reports to the controller that the module has a new task. After the controller confirms the module has been set to `enabled` mode, it adds back the read/write permissions and marks the module as "busy". If the module was marked as "busy", then the permissions are granted immediately.

When the timer interrupt occurs, for the modules that are marked as "idle", the monitor reports to the controller the module has no pending task. At the end of the timer interrupt service routine, all modules are marked as "idle", and permissions of memory pages of all modules are removed.

The controller part calls the PM callbacks provided by the driver to set the device to `enabled`/`disabled` mode based on the inference of the monitor. It confirms the monitor after a module has been set to `enabled` mode.

5.4 Evaluation

We evaluate the central PM agent on Pandaboard Rev B2 which uses OMAP4460 SoC. It runs Linaro Android release 13.10. The kernel version is 3.2.

Effectiveness of the central PM agent

We show that the central PM agent provides power management to drivers lacking PM. To demonstrate efficacy, we show that our power management decisions match those of device drivers with PM.

We use the central PM agent to automatically control the SDIO controller and I2C controller. The WiFi NIC on Pandaboard uses them to communicate with CPU. The

SDIO driver does not have a good power management policy, while the I2C does have one. We replace power management code in I2C driver with some dummy code, which is only used to calculate how much time the original power management code allows the device to stay in `disabled` mode. We compare the result with when it is controlled by our central PM agent. We connect the Pandaboard to WiFi and intensively browse websites using the default Android browser for 10 min. The result shows for I2C, the device driver would have allowed I2C to stay in `disabled` mode for 131.6 s. Our central PM agent allows it to do so for 126.1 s, which is a 4.2% decrease. The decrease is due to the monitor has the latency when inferring there is no pending task. The central PM agent sets the SDIO in `disabled` mode for 42.2 s.

We also use the central PM agent to automatically control the MMC controller. The file system of Pandaboard is on a MMC card. The MMC driver has power management implementations. In 10 min, we transmit four 0.2MB files to Pandaboard and call "sync" after each transfer every minute. We also open some Android apps once every minute, which causes the OS to load programs from the MMC card. The result shows the device driver would have allowed MMC to stay in `disabled` mode for 498.6 s. Our central PM agent allows it to do so for 493.3 s, which is a 1.0% decrease.

With the central PM agent controlling the above three modules, the Android device is able to run robustly for more than one day. We haven't seen any loss of functionality caused by the central PM agent.

Overhead of the central PM agent

The overhead mainly comes from the memory exception handling. We measure this overhead using hardware performance counters. It shows handling each memory exception costs around 2500 cycles, which is around 8 µs when the CPU runs at its lowest frequency 300 MHz. Note this overhead occurs only once for each module every $d = 100$ ms. Even for the busiest driver, adding 8 µs extra execution time every 100 ms is negligible.

5.5 Hardware support

The monitor part of our central PM agent implementation has the following limitations: (*i*) it relies on memory exceptions, which adds overhead, though small; (*ii*) it is unable to detect whether a slave module has pending tasks if it is used by a computational module other than CPU; (*iii*) it only works for modules that frequently communicate with the CPU through register access. We next show that small hardware modifications can completely eliminate these limitations.

First, a hardware module essentially knows if it is busy processing a task. Today, some but not all SoC components provide a register bit indicating this information. If all SoC modules provide a *busy/idle register*, the monitor can detect when the modules have finished all pending tasks by periodically sampling these registers; this eliminates the overhead caused by memory exceptions. Note that we propose sampling the busy/idle register, rather than letting the module interrupt the central PM agent every time it finishes a task; the sampling approach filters out the transient idle period in the middle of consecutive tasks, preventing excessive power state transitions.

Second, the hardware global power manager on today's mobile SoCs is already capable of detecting when there is a new task for a module that is `disabled`, no matter whether the task is issued by the CPU or other computational modules. For example, OMAP4's global power manager, called the Power, Reset and Clock Management module (PRCM), listens on the system bus and detects when there is bus activity targeted at a module that is `disabled` [18]. This is how it automatically reactivates bus interface clock(§2.2). We only need to modify the PRCM to let it expose this information to the central PM agent.

With the above hardware support, the monitor part of the central PM agent is able to monitor all on-chip modules with little overhead, thus solving the limitations.

6 Related work

Android opportunistically suspends the entire system if there is no user interaction for a period of time. It allows an app to change the power management policy through the wakelock framework. That is, an app can hold a wakelock to prevent the suspension. The opportunistic suspension and the wakelock framework together create burden on application developers and lead to power bugs [19, 20].

Our work focuses on Linux's runtime power management, which is complementary to Android's opportunistic suspension. Compared to the opportunistic suspension approach, runtime power management is more fine-grain in that it allows individual SoC modules to enter a low-power state when they are not used. With careful runtime power management, if all modules are idle, SoCs can reach similar power states as system-wide suspension [21]. Runtime PM will become more important in the future as there will be more always-on services, e.g., voice recognition and live image processing [22].

There is a large body of literature on power management policies and their system realization, e.g., [15–17]. Our work is orthogonal since we address where in the system power management should be implemented.

7 Conclusion

We argue that device drivers are not the best place to implement power management. Because due to the dependencies in the SoC power management hardware, one device driver that does not have good power management ruins the efforts of other drivers and we present empirical evidence that device driver developers are not doing a good job in implementing power management. We show the information needed for good power management policies is available outside the device drivers and implement a software central PM agent to control three modules on OMAP4 SoC. The central PM agent does as efficient power management as the drivers do. It is also able to manage a module (SDIO) that does not have power management implemention in its driver. As future work, we propse to add hardware support to the central PM agent to eliminate the limitations of a pure software implementation.

Acknowledgement

The work was supported in part by NSF Awards #1054693, #1065506, and #1218041. The authors thank the anonymous reviewers for their useful feedbacks.

References

[1] Govindraj Raja. Omap2+: Uart: Add runtime pm support for omap-serial driver. http://www.spinics.net/lists/linux-omap/msg58443.html, 2011.

[2] Mark Brown. spi/s3c64xx: Implement runtime PM support. http://www.spinics.net/lists/linux-samsung-soc/msg08912.html, 2012.

[3] Roger Quadros. USB: Implement runtime idling and remote wakeup for OMAP EHCI controller. https://lkml.org/lkml/2013/7/10/355, 2013.

[4] Paul Walmsley. Watchdog: omap_wdt: add fine grain runtime-pm. http://permalink.gmane.org/gmane.linux.ports.arm.omap/54608, 2011.

[5] Felipe Balbi. gpio: omap: be more aggressive with pm_runtime. http://www.spinics.net/lists/linux-omap/msg64196.html, 2012.

[6] OMAP4460 multimedia device technical reference manual, 2011.

[7] NVIDIA Tegra 4 4-PLUS-1 quad-core processors technical reference manual, 2013.

[8] Samsung Exynos 5 quad (exynos 5250) RISC microprocessor user's manual, 2012.

[9] i.MX 6Dual/6Quad applications processor reference manual, 2013.

[10] Intel Atom processor Z36xxx and Z37xxx series datasheet, 2013.

[11] P. Choudhary and D. Marculescu. Power management of voltage/frequency island-based systems using hardware-based methods. *IEEE Trans. VLSI Systems*, March 2009.

[12] T. Hattori, T. lrita, M. Ito, E. Yamamoto, H. Kato, G. Sado, Y. Yamada, K. Nishiyama, H. Yagi, T. Koike, Y. Tsuchihashi, M. Higashida, H. Asano, I. Hayashibara, K. Tatezawa, Y. Shimazaki, N. Morino, K. Hirose, S. Tamaki, S. Yoshioka, R. Tsuchihashi, N. Arai, T. Akiyama, and K. Ohno. A power management scheme controlling 20 power domains for a single-chip mobile processor. In *ISSCC, 2006*, pages 2210–2219, Feb 2006.

[13] Mark Brown. Input: samsung-keypad: Implement runtime power management support. http://www.spinics.net/lists/linux-input/msg18796.html, 2011.

[14] Vivek Gautam. dwc3/xhci: Enable runtime power management. https://lkml.org/lkml/2013/1/28/229, 2013.

[15] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM TECS*, (3), 2003.

[16] Tajana Simunic, Giovanni De Micheli, and Luca Benini. Event-driven power management of portable systems. In *Proc. Int. Symp. System Synthesis*, 1999.

[17] Qinru Qiu and Massoud Pedram. Dynamic power management based on continuous-time markov decision processes. In *Proc. ACM/IEEE DAC*, 1999.

[18] Christophe Vatinel. OCP disconnect proposal. http://www.ocpip.org/uploads/documents/OCP_Disconnect_Proposal_0.3.pdf, 2008.

[19] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. ACM MobiSys*, 2012.

[20] Abhilash Jindal, Abhinav Pathak, Y Charlie Hu, and Samuel Midkiff. Hypnos: understanding and treating sleep conflicts in smartphones. In *Proc. ACM EuroSys*, 2013.

[21] Rafael J Wysocki. Technical background of the android suspend blockers controversy, 2010.

[22] Rafael J Wysocki. Power management in the Linux kernel: current status and future. http://events.linuxfoundation.org/sites/events/files/slides/kernel_PM_plain.pdf, 2013.