Technical Background of the Android Suspend Blockers Controversy

Rafael J. Wysocki
rjw@sisk.pl
Faculty of Physics, University of Warsaw / SUSE Labs, Novell Inc.

November 12, 2010

Abstract

This article investigates technical issues behind the discussion that resulted from a recent attempt to merge the Google Android's opportunistic suspend feature into the mainline Linux kernel. It first describes the opportunistic suspend framework and the problems that it is intended to address. Next, it discusses difficulties related to using full system suspend for aggressive power management and explores alternative solutions that do not introduce them. Finally, it presents mainline kernel work targeted at addressing the problems that originally caused the opportunistic suspend feature to be developed, which is less intrusive and allows of more flexibility.

1 Introduction

If you have been following the Linux kernel development for a few recent months, it has been hard to overlook the massive thread on the Linux Kernel Mailing List (LKML) resulting from an attempt to merge the Google Android's suspend blockers framework into the main kernel tree [1]. As probably everyone interested in Linux knows already, the suspend blockers patches from Google have been rejected [2], but the reason for that doesn't seem to be particularly clear to anyone who's not deeply engaged in the kernel development. Arguably, the presentation of the patches might be better and the explanation of the problems they addressed might be more straightforward [3], but in the end it appears that merging them wouldn't be the smartest thing from the technical point of view. Unfortunately, though, it is difficult to explain that without describing the technical issues behind the suspend blockers patchset.

In fact, suspend blockers, or *wakelocks* in the original Android terminology, are a part of a specific approach to power management, which is based on aggressive utilization of full system suspend to save as much energy as reasonably possible. In this approach the natural state of the system is a sleep state [4], in which energy is only used for refreshing memory and providing power to a few devices that can generate signals to wake the system up from that state. The working state, in which the CPUs are executing instructions and the system is generally doing some useful work, is only entered in response to a wakeup signal from one of the selected devices. The system stays in that state only as long as necessary to do certain work requested by the user. When the work has been completed, the system automatically goes back to the sleep state.

Generally, this sounds like a good idea. Still, as it often happens, the final outcome depends on the details and it turns out that the Android's implementation of the approach outlined above, discussed in Section 2, leads to problems that are quite hard to solve. Moreover, if this approach is used on a generic GNU/Linux system, like an x86 PC, it will potentially cause some timekeeping issues to appear, as explained in Section 3. For these reasons, the question arises if there is an alternative way to achieve comparable energy savings. The general answer to it is yes, there is one, based on the *cpuidle* framework [5], but it also has some disadvantages, as shown in Section 4.

Although the way in which Android uses full system suspend for aggressive power management is controversial, it doesn't seem reasonable to dismiss the idea of automatic suspend in general as a valid method of saving energy. Other operating systems actively use it, which allows them to claim an advantage over GNU/Linux in the power management field [5]. Some Linux-based systems, like the OLPC project [6], also use it successfully and they have to deal with the same problems that Android wakelocks are supposed to address. However, on these systems full system suspend is always initiated by user space which allows of more flexibility and requires some problems addressed with the help of wakelocks on Android to be handled in a different way.

From the kernel's point of view these systems are really quite similar to systems that only use full suspend occasionally. All of them have to avoid race conditions between system suspend and system wakeup events, which is one of the problems addressed by the Android wakelocks framework. Yet, the Android's solution only applies to systems that use suspend aggressively and initiate it from kernel space, so it doesn't really cover, for example, the systems that only suspend at the user's request. On the other hand, if there's a way to avoid the race conditions between system suspend and system wakeup events on those systems, it will also apply to the systems that use full system suspend aggressively, but start it from user space, like OLPC. That practically has to be implemented with the help of a framework which is similar to the Android's wakelocks, but which is different enough to be substantially less controversial. There is some effort in that direction under way, which resulted in an initial patch merged into the main kernel tree a couple of months ago [7]. The new interface introduced by it and the general idea of the new framework are presented in Section 5. There is hope that Android might use this framework in the future and either add a small out-of-the-tree patch introducing a kernel-based suspend mechanism on top of it, or switch to the more popular model in which full system suspend is always started from user space.

2 Opportunistic Suspend the Android's Way

The Google's Android operating system has been designed with mobile devices, like cell phones or tablets, in mind. Such devices are generally not expected to be able to carry out CPU-intensive computations or do very I/O-intensive kind of work like workstations or servers. Thus, although they are supposed to allow their users to preform some tasks that can also be done on a desktop computer, like browsing the Web, reading and sending e-mail, listening to music or playing simple video games, they generally need not support extensive multitasking and they are inherently single-user. Generally speaking, their performance requirements are quite different from the performance requirements of other Linux-based systems.

On the other hand, it is essential that these devices can be used for as much time as possible without connecting to any external power sources (e.g. AC power). In other words, their *battery life* has to be as long as reasonably possible, even at the expense of performance.

In principle there are multiple ways to achieve this goal. In particular, one can use various *runtime power management* (runtime PM) techniques for this purpose [8]. Basically, hardware components of the system may be powered down when they are not used and powered up when they are needed again. For example, the display is usually one of the most power-hungry components of a hand-held device like a mobile phone, so it makes sense to switch it off when the user is not looking at it[1]. It also is generally possible to reduce the capacity of various system components if full capacity is not necessary. For instance, the display can be dimmed when the ambient light is not too bright, so that it doesn't draw too much power in vain. If all of the system's hardware components are treated this way, they all will be powered down when the system is completely idle and, in theory, from the energy usage point of view, it will behave as though it has been suspended.

Android uses runtime PM to some extent, but its developers claim that runtime PM alone is not sufficient to get acceptable battery life of an Android-based system in the majority of cases [9]. It is difficult, though, to either universally accept this claim, or reject it entirely, because its validity depends on a number of different factors. For instance, it depends on the particular implementation of runtime PM in use as well as on the capabilities of the hardware in question. It also depends on what length of the system's battery life is regarded as "acceptable".

The implementation of runtime PM certainly is important. Among other things, runtime PM can be implemented as either more aggressive, or more performance-oriented and most likely the expected energy savings in both cases will be different. Moreover, the heuristics used to decide whether or not a particular hardware component can be powered down at a given instant of time play a significant role, but there is no golden rule allowing one to choose the most appropriate heuristics for every possible situation. It also is generally difficult to incorporate those heuristics into the drivers handling the hardware components without hampering their basic functionality. In addition to that, hardware components may depend on one another in various ways that need to be taken into account (e.g. if there's a bus with multiple I/O devices on it and a bridge connecting it to the rest of the system, then the bridge cannot be powered down if any I/O devices on the bus are in use). There is the I/O runtime PM framework in the Linux kernel introduced to help driver writers deal with these difficulties [8], but the first version of Android had been released before that framework was developed. In consequence, to the best knowledge of yours truly, none of the versions of Android available to date uses the kernel's I/O runtime PM framework in any significant way. Therefore one

[1] Of course, this is not always easy to tell, but some heuristics can be applied to figure that out more or less accurately

has to believe that the Android's implementation of runtime PM cannot be improved so that it's possible to achieve better battery life of devices running it.

The capabilities of the hardware the operating system has to run on matter as well, because on many types of hardware the level of energy savings attainable by suspending the system cannot be reached with the help of runtime PM. That can be explained on the example of PCI hardware[2]. Namely, PCI devices, or more precisely PCI functions[3], support up to four different modes of operation called *power states*. There is the *full-power state*, labeled as $D0$, in which the device can process I/O normally, and there are *low-power states*, labeled as $D1$–$D3$, in which I/O cannot be processed normally, but substantially less power is drawn than in the full-power state [11]. The low-power states differ from each other in the amount of power drawn by the device and the time needed to switch it from $D0$ to the given low-power state and back. $D1$ is the narrowest low-power state, in which the device draws more power than in the other low-power states, but can be switched back to $D0$ in the shortest time. $D3$, in turn, is the deepest low-power state. All PCI devices are required to support the full-power state, $D0$ (quite obviously), and the deepest low-power state, $D3$, while the intermediate low-power states $D1$ and $D2$ are optional[4]. A PCI device can be programmed to switch from $D0$ to one of the supported low-power states and, if it already is in a low-power state, it can be programmed to switch back to $D0$[5]. This means that even if the device is in $D3$, it is still drawing power, at least as much as necessary to switch back into the full-power state[6]. However, it is possible to remove power entirely from a PCI device[7] and that's where the hardware-related difference between full system suspend and runtime PM originates (as far as PCI hardware is concerned).

PCI devices cannot be programmed to cut off power from themselves, so this has to be done from the outside of the device in question. If the device is on a bus segment (e.g. in a PCI slot), power can be removed entirely from that bus segment by programming its parent bridge to switch to $D3$[8]. Of course, as a result power will not be provided to any devices on that bus segment until the bridge is programmed to switch back to $D0$. In turn, if the device is not on a separate bus segment (e.g. it may be integrated into the computer's chipset, in which case it is called a planar device), the only way to remove power from it is to turn off the circuitry providing it with power, sometimes referred to as the device's *power resources*. This can be done directly if the system's hardware architecture is well known or its firmware provides the kernel with a suitable interface, but in many cases it only is done automatically by the system's firmware when the whole system is going to be powered off or suspended. More specifically, on many systems there is a firmware interface that the kernel is supposed to use in the last phase of system shutdown, hibernation and suspend to ask the firmware to turn off all power resources, except for a few selected ones, needed to support wakeup signaling. There may be no other way to turn the power resources off, in which case it is necessary to shut down, hibernate or suspend the system to remove power entirely from a number of devices. In that case full system

[2]Contemporary Android-based systems usually don't contain standard PCI hardware, but other types of hardware tend to have similar properties.

[3]The idea is that one physical piece of hardware occupying a PCI slot can consist of multiple I/O modules which can be programmed and used more-or-less independently of each other. These I/O modules are then referred to as *functions* by the official PCI documentation [10], but since they are the hardware units that process I/O, they are I/O *devices* from the kernel's point of view. For this reason they are referred to as devices in what follows.

[4]For this reason and since $D3$ is more attractive from the energy saving point of view, $D1$ and $D2$ are used very rarely. The Linux kernel's PCI subsystem doesn't use them at all.

[5]If $D1$ and $D2$ are supported, the device can be programmed to switch from $D1$ to $D2$ and to switch from $D1$ or $D2$ to $D3$, but it cannot be programmed to switch from $D3$ to $D1$ or $D2$, or to switch from $D2$ to $D1$.

[6]In fact, every PCI function contains a set of standard registers, referred to as *standard configuration registers*, whose contents are preserved in low-power states [11], so even in the deepest low-power state it has to draw as much power as necessary to preserve the contents of all these registers.

[7]This observation led to the most confusing part of the PCI power management terminology. Namely, one can think that a device with cut off power is in a special "low-power state" and in that situation it "behaves" as though it were in $D3$, except that it cannot be programmed to do anything. Hence, the designers of PCI power management interface decided to treat that "state" as a variant of the deepest low-power state available to PCI devices and labeled it as $D3_{cold}$. In consequence, the "normal" $D3$, in which the device can be programmed to switch to $D0$, was called "software accessible $D3$" and labeled as $D3_{hot}$. In my not so humble opinion this naming convention doesn't make sense for two reasons. First, PCI devices cannot be programmed to switch into $D3_{cold}$, they only can end up in that "state" as a result of an external action (e.g. removing power from the bus segment they are on). Second, if a PCI device is in $D3_{cold}$, it doesn't automatically go into $D0$ after power has been restored [11]. Therefore $D3_{cold}$ is fundamentally different from $D3_{hot}$ and they shouldn't be regarded as two "variants" of the same low-power state. Consequently, throughout the present article the symbol $D3$ is used to refer to the state called $D3_{hot}$ in the official PCI documentation and the $D3_{cold}$ "state" is regarded as a separate special case. Accordingly, the symbols $D3_{hot}$ and $D3_{cold}$ are not used.

[8]In PCI terminology there are low-power states of bus segments. In particular, the low-power state of a bus segment in which power is entirely removed from it is labeled as $B3$ and the bus segment is switched to it by programming its parent bridge to switch to $D3$.

suspend will always save more energy than runtime PM[9].

Apparently, this was the case on the Android's first reference platform, the Google G1 [12], which must have influenced its creators' power management philosophy. Since at that time the kernel did not support I/O runtime PM at the core level and on the given hardware platform runtime PM couldn't save as much energy as full system suspend anyway, they decided to use an approach based on suspending the entire system as frequently as reasonably possible. That, however, required them to address a number of issues.

First, they noticed that there were races between the system suspend process and wakeup events. Namely, on a typical Linux-based system, the suspend process is started by writing `mem` to the `/sys/power/state` file on *sysfs*. One of the first things it does is to freeze user space processes (except for itself) and after that's been completed user space cannot react to any events signaled by the kernel. Consequently, if a system wakeup event occurs exactly at the time `/sys/power/state` is written to, user space may be frozen before it will have a chance to consume the event, which will be delivered to it only after the system is woken up from the sleep state as a result of *another* wakeup event. Unfortunately, on a cell phone the "deferred" wakeup event may be a very important incoming call, so the above scenario is hardly acceptable for this type of devices.

This issue was addressed with the help of wakelocks. Essentially, a wakelock is an object that can be in one of two states, active or inactive, and the system cannot be suspended if at least one wakelock in it is active. Thus, if the kernel subsystem handling a wakeup event activates a wakelock right after the event has been signaled and deactivates it after the event has been passed to user space, the race described in the previous paragraph can be avoided. For this purpose, however, one has to specify how exactly system suspend is going to be prevented from happening if there is an active wakelock. Of course, if the wakelock is activated when the suspend process is already in progress, the only way to prevent the system from suspending is to abort the suspend process, but if the wakelock has already been active before the suspend process is started, there are a few different possibilities. For example, the operation of writing `mem` to `/sys/power/state` may fail if there's an active wakelock or it may block until there are no more active wakelocks in the system. Whichever option is chosen, there still is a problem, because user space may be frozen after the kernel has deactivated all wakelocks, but before the handling of wakeup events in user space is complete.

For instance, consider a keyboard that also is a source of wakeup events, so if the system is in a sleep state and one of the keys is pressed on the keyboard, the system will be woken up. In that case it may not make sense to suspend the system if any keys have just been pressed. More precisely, it may not make sense to suspend it until user space can read the key codes and react to them as appropriate. Thus there should be a mechanism allowing the processes that read the key codes from the kernel to prevent `/sys/power/state` from being written to until they finish the work. Of course, they cannot prevent arbitrary privileged processes from writing to `/sys/power/state`. However, if there is one special process, a *power manager*, attempting to write to `/sys/power/state` whenever it decides that it's desirable to suspend the system, then in theory the key code readers may be synchronized with it by means of inter-process communication (IPC) of some sort.

Assume that every time a key code is read from the kernel, the user space process that read it uses an IPC mechanism to prevent the power manager from writing to `/sys/power/state`. Then, system suspend won't be started as long as user space key code readers are doing their work, but unfortunately that doesn't prevent a key press event from slipping through. Namely, if the user is typing a message, there may be random delays between consecutive key press events. When one key is pressed, the keyboard driver promptly detects that and activates a wakelock. The key code is registered and the user space process waiting for it is woken up. This process activates the IPC preventing the power manager from writing to `/sys/power/state` and reads the key code from the kernel. Then, the keyboard driver deactivates the wakelock and waits for another key press event. In the meantime, the process that read the key code does whatever it is supposed to do after reading it and uses the IPC to let the power manager know that it can proceed with suspending. As a result, the power manager may decide to write to `/sys/power/state` and if there are no active wakelocks in the kernel at that time, the suspend process will be started. Now suppose that another key is pressed exactly at that time, the wakelock is activated and the key code is read by user space before the suspend process starts freezing it. The keyboard driver deactivates the wakelock after the key code has been read and the suspend process need not notice the activation and deactivation of the wakelock, in which case it will continue as though nothing happened. Of course, the user space key code reader hasn't done its work yet, but the IPC used to prevent the power manager from writing to `/sys/power/state` won't be effective, because this operation has been carried out already. In consequence, the reader process will be frozen along with the rest of user space and the key code will be processed after the subsequent system resume resulting from another wakeup event.

There are a few ways to prevent the above scenario from taking place. For example, one may intro-

[9]The difference need not be substantial, though, because the majority of PCI devices in $D3$ draw only a little power and this amount need not matter at all, depending on the system designer's expectations.

duce a mechanism by which system suspend will always be aborted if any wakelocks are activated after `/sys/power/state` has been written to, even if these wakelocks are immediately deactivated. However, the Android developers apparently thought that would be wasteful (the user space key code reader may actually manage to handle the event entirely before being frozen) and instead they introduced an API allowing user space to create, activate, deactivate and destroy wakelocks. Thus, on Android, instead of using IPC to prevent a power manager from writing to `/sys/power/state`, the user space key code reader will activate a wakelock that will be deactivated after dealing with the key code entirely.

Of course, this allows the races between wakeup events and the system suspend process to be avoided completely, but at the same time it is somewhat controversial, because it appears to violate the principle of separation between the kernel and user space. After all, in this approach user space is allowed to manipulate objects inside the kernel almost directly. Still, if that only affected an operation started by user space, which system suspend was on typical Linux-based systems, that wouldn't be much of a deal. Unfortunately, on Android this is not the case because of the way it addresses the second issue related to the aggressive utilization of system suspend.

That issue is, basically, when exactly system suspend should be started or, in other words, what conditions should cause the system to be suspended. Although there are no simple answers to this question in general [13], it should be clear that if system suspend is to be used aggressively, it ought to happen as often as reasonably possible. Of course, the system cannot be suspended when it is doing useful work, but in principle it may be suspended as soon as the work is done. Thus one can think that whenever the system is not doing useful work, there is an *opportunity* to suspend it. There are systems which aggressively try to use that opportunity and for this reason they can be referred to as systems that suspend *opportunistically*. Evidently, Android is one of them.

For opportunistic suspend to work one needs to have reliable criteria for deciding whether or not the system is doing something useful. The criterion used by Android is very simple and straightforward: The system is suspended whenever there are no active wakelocks. More precisely, every deactivation of the last active wakelock starts the system suspend process. Attempts to suspend that would certainly fail are avoided this way. Yet, it requires every user space process that does important, or useful, work to use wakelocks, which adds unusual and cumbersome issues for application developers to deal with [14]. In turn, the processes using wakelocks can impact the system's battery life quite significantly, so the ability to use wakelocks has to be regarded as a privilege that should not be given unwittingly to all applications. Unfortunately, however, there is no general principle the system designer can rely on to figure out what applications will be important enough to the system user to allow them to use wakelocks by default, so ultimately the decision is left to the user. This, of course, is only going to really work if the user is qualified enough to make the decision, which quite obviously need not be the case. That problem is discussed in more detail in Section 3.

Since the Android developers chose to use the deactivation of the last active wakelock as the suspend-triggering condition, they also decided that it would be better to start the suspend process from kernel space. This appeared to be logical, because otherwise a power manager in user space would only be necessary for writing to `/sys/power/state` whenever the kernel ordered it to do that. Consequently, in the Android kernel there is a work item whose job is to start the suspend process when the last active wakelock is deactivated. After doing that, its work function reschedules the execution of itself. Thus on Android the suspend process runs in the context of the workqueue thread that happens to execute this work function, which brings along the violation of the separation between the kernel and user space mentioned earlier. Namely, it causes the (almost) direct manipulation of wakelocks, being kernel objects, by user space to influence decisions made by the kernel in a way that may hinder the system's expected functionality (i.e. as a result of it the battery may be exhausted much faster than expected resulting in a denial of service and the kernel has no means to prevent that from happening). Although this problem doesn't seem to be particularly serious from the practical point of view, there are other issues related to the Android's implementation of opportunistic suspend that directly affect user experience.

3 Problems with Kernel-Based Opportunistic Suspend

To recap, Android uses opportunistic suspend as one of its most important power management measures. It implements this feature in such a way that the suspend process is started from kernel space whenever there are no active wakelocks. Moreover, wakelocks are used for avoiding races between the suspend process and wakeup events, as described in Section 2. User space is allowed to manipulate wakelocks, although they formally belong to the kernel, and there is a special interface between the kernel and user space designed specifically for this purpose.

It turns out that the Android's kernel-based implementation of opportunistic suspend leads to a number

of issues that in part are purely theoretical and in part manifest themselves in practice, resulting in some undesirable effects which are not too difficult to observe on Android-powered devices.

First, there is the problem of broken separation between the kernel and user space mentioned in the last paragraph of Section 2. Although it is rather theoretical, it is closely related to the much more practical observation that applications allowed to use wakelocks can do quite a bit of damage by causing the battery to be drained unexpectedly quickly. For this reason the ability to use wakelocks has to be regarded as a privilege that should be granted very carefully. In some, arguably rare, cases there may be even security implications of granting it to a wrong application[10]. Therefore someone has to decide what applications should be granted this privilege, but that decision cannot be made in advance by the system designer, because only the user is able to determine the set of sufficiently important applications. However, if the user is expected to make such a decision, he (or she) should be informed *exactly* about the possible consequences of it. Moreover, the user should be able to disallow chosen applications the use of wakelocks at any time. On Android, though, that simply doesn't happen[11].

An Android system contains a few applications allowed to use wakelocks by default, like *GMail*, and the user cannot change that. In turn, for applications installed by the user, the permission to use wakelocks has to be granted *before* installing them or they won't be installed at all. Once an application that wants to use wakelocks has been installed, it will use them and the user won't be able to prevent it from doing that in any other way than by uninstalling it. Thus a knowledgeable user only really has the choice to either allow the application to ruin the device's battery life if it wants to, or to restrain himself from installing or using the application at all. Needless to say, that may not be an attractive choice for many users, especially given that one of the applications in question is *Google Maps* [12].

In addition to that the vast majority of Android users have no idea about the opportunistic suspend feature and how it works, so they really don't have sufficient information to make reasonable decisions in this respect. Worse yet, when installing a new application the user is shown a list of "permissions" the application needs to function properly and the permission to use wakelocks is present in this list as a part of the "System tools" item under the label "prevent phone from sleeping"[13]. There is no indication whatsoever that this basically translates into "this application wants to be able to make the phone drain battery faster", so in the majority of cases the users don't really know what they are agreeing to. Moreover, even if they knew, they probably would agree to install the applications anyway, because they wanted to use them in the first place, at least occasionally.

Fortunately, an Android user can check which running applications prevent the device from suspending most of the time[14] and can forcibly stop them if they need not run at the moment. Still, this information is generated on the basis of wakelock statistics collected by the kernel, consisting of several numbers having different meanings per wakelock, but it is shown to the user in the form of a single meter (i.e. a percentage represented by a "usage bar"). It can't be accurate for this reason, although it generally gives the user a good idea of who's responsible for draining the battery in the first place. Once identified, an offending application has to be forcibly stopped every time it has run for a while and doesn't need to run any longer, which is cumbersome and requires the user to remember to stop it. Definitely, this is not the most convenient mode of operations one can imagine. Perhaps if the Android user space had been designed differently, it would have been easier to handle this issue in a more convenient way, but it seems that the Android's use of opportunistic suspend really is the source of the problem.

Another issue with it is that some advertised features of applications don't really work because of it. Namely, some applications (e.g. *GMail*, the e-mail application, news and weather applications, etc.) are supposed to periodically check things on remote Internet servers. The e-mail programs check if there are any

[10] After all, *availability* is generally regarded as one of the key principles of information security [15] and unexpected draining the system's battery is nothing else than a violation of availability.

[11] At least I was unable to find such an interface on a Nexus One phone with Android 2.2.

[12] My experience indicates that *Google Maps* can actually hurt an Android phone's battery life quite a bit. This year in Boston I used it before leaving the hotel room to find a way to go to some other place. I downloaded the map and walking directions into the phone and checked them a few times on my way. When I finally got to the destination, I pressed the phone's "turn off the screen" button and put it into a pocket, leaving *Google Maps* in the foreground. Then, I had been moving from one place to another relatively quickly for a couple of hours. When I got back to the hotel, I realized that the phone's battery was almost empty, although I had charged it before leaving the room. The battery got drained almost entirely during my trip, even though I didn't make or receive any calls and in fact I didn't even look at the phone's screen! Apparently, *Google Maps* used the GPS all the time when I was moving to update my current location and it was using wakelocks to block the opportunistic suspend mechanism, so the device burned much more energy than expected.

[13] Along with some other things like "change Wi-Fi state", whatever that means.

[14] On a Nexus One phone with Android 2.2 this information is available from the "Battery use" item in the "About phone" menu under "Settings".

new messages to read, the weather applications check for new weather forecasts and so on. For this to work they need to run when they are supposed to do their checks, but they obviously aren't running when the system is in a sleep state. They also need to access the network which is unavailable after the system has been suspended. Thus the periodic checks the applications are supposed to make aren't really made at that time. In fact, they are only made when the system is in the working state incidentally for another reason and there happens to be the time to do the checks[15]. This most likely is not what the users of the affected applications would expect to take place[16], but making it work as expected is not as easy as one might think either.

If the system is in a sleep state when there is time for an application to do some work, the only way to allow it to do that work is to wake the system up. In particular, if the instant of time when the application is going to start doing the work is known in advance, one can use a clock-generated wakeup event to wake up the system right before that time. Every hardware platform known to yours truly has a real time clock (RTC) device being essentially a persistent clock that is running even after the system has been suspended or powered off and that usually can be used exactly for this purpose. Most often the RTC can be programmed to generate a wakeup event at a specific instant of time. This mechanism is referred to as the *RTC wake alarm* and is available through the `/sys/class/rtc/rtcN/wakealarm` *sysfs* interface, where `N` is the number of the RTC device within the system, starting from 0 [16]. Although that generally is straightforward and simple, up to some gory technical details you don't really want to know unless you're writing a driver for an RTC device, there is one difficulty related to it: When the alarm time expires, the RTC wake alarm has to be set once again from scratch. Moreover, when it is set to a new value, the previous setting is simply forgotten, so if there's a need to set it to trigger at two different instants of time, one has to wait until it triggers at the earlier instant and then set it once again to trigger at the later one. This complicates things quite a bit when there are two different applications in the system that need to do some work periodically with different frequencies.

To illustrate this complication suppose that there is an e-mail client configured to check for new e-mail every 15 minutes and a news application set up to look for a news update every 20 minutes. Let T_0 be the time the e-mail client is started and suppose that the news application starts 2 minutes later. Then, if the RTC wake alarm is to be set by each application individually, the e-mail client will set it to $T_0 + 15$ minutes, but then the news application will change this setting to $T_0 + 22$ minutes. In consequence, if the system is suspended, for example, at the time $T_0 + 10$ minutes, the RTC wake alarm will not trigger at the time the e-mail client should do its check, it will only trigger at the time set by the news application.

To address this problem one can use a list of instants of time when the RTC wake alarm should trigger, but this list has to be managed somehow. Specifically, a new RTC wake alarm time always has to be set right after the previous one has expired, so there should be a guarantee that at least one process will try to do that every time. The seemingly simplest way to achieve this goal is to introduce a special process that will repeatedly set the wake alarm and put itself to sleep to wake up right after the alarm has triggered[17]. Then, the applications that need to do work periodically may communicate with this process using some form of IPC to tell it what instants of time the wake alarm should trigger at. This, however, is dangerously close to the user space power manager idea discussed in Section 2 and the question arises whether it may be better to implement a power manager in user space and let it take care of the issue at hand, among other things.

There is one more problem with full system suspend that is related to time measurements. Unfortunately, though, it is not limited to the opportunistic system suspend initiated from kernel space. Namely, every suspend-resume cycle, regardless of the way it is initiated, introduces inaccuracies into the kernel's timekeeping subsystem. Basically, the kernel's timekeeping subsystem relies on two types of resources, *clock sources* and *clock event devices* [17] and they both are affected by system suspend and resume.

A clock source is a kernel object associated with and representing a piece of hardware that can be used to measure the flow of time, like the High Precision Event Timer (HPET) or the Time Stamp Counter (TSC) CPU register [18]. Usually, when the system goes into a sleep state like ACPI[18] $S3$ (memory sleep) or ACPI

[15]This is my observation from a Nexus One phone with Android 2.2.

[16]When I configure an e-mail client application to check for new e-mail every 15 minutes, I expect it to actually do that every 15 minutes and not "every 15 minutes or less often depending on whether or not the system automatically suspends in the meantime", even it that application is running in the background.

[17]On Android it would also have to use a wakelock to prevent the system from being suspended before the new wake alarm time is set. Moreover, the wakelock would have to be kept active for some time to allow the "periodic" applications to run and presumably activate their own wakelocks.

[18]The Advanced Configuration and Power Interface (ACPI) specification defines an interface between the platform firmware (e.g. the BIOS on a PC) and the operating system for configuring platform-dependent hardware and changing its power states. Among other things, it introduces system states $S0$–$S4$, where $S0$ is the working state and $S1$–$S4$ are sleep states. The most widely used ACPI sleep states are $S3$ and $S4$, corresponding to suspend to RAM and hibernation, respectively.

$S4$ (hibernation), the clock source's hardware is powered off, so it has to be reinitialized (typically from scratch) during system resume. For this reason, the global kernel variables representing the current time, `xtime` and `wall_to_monotonic`[19], need to be readjusted every time during system resume to keep track of the time spent in the sleep state[20]. This is done with the help of the `read_persistent_clock()` function, the definition of which is unfortunately architecture-specific and, for example, on x86 its result is rounded down to the closest 1 second boundary[21]. In consequence, on x86, if the suspend time is saved exactly at $1\,s$ boundary, the systems sleeps for about $2\,s$ and `read_persistent_clock()` is called (during resume) exactly $2.3\,s$ after the suspend time has been saved, only $2\,s$ will be added to `xtime` and subtracted from `wall_to_monotonic`[22]. The remaining $0.3\,s$ will be effectively lost, although it would have been taken into account if the suspend-resume cycle hadn't occur. On x86 this mechanism introduces a random shift up to $1\,s$ between the persistent clock and the kernel monotonic clock in every suspend-resume cycle[23]. Likewise, on x86 every measurement of a time interval across a suspend-resume cycle is subject to a random error of the order of $1\,s$. Moreover, kernel timers are affected in a similar way.

Kernel timers are objects used for scheduling future execution of specific code. Each timer contains, among other things, a pointer to a function to run in the future and a representation of the desired time of its execution. Depending on the type of the timer, that representation may be either the value of the `jiffies` variable corresponding to the instant of time the function is supposed to be run at, or the kernel's monotonic clock value corresponding to that instant of time, as returned by function `ktime_get()`[24]. In both cases this number is used to program a hardware timer to trigger an interrupt on the local CPU at the desired time.

The hardware timers used for this purpose are referred to as *clock event devices* in the kernel terminology [17] and usually there is at least one of them per CPU core[25]. Typically, they each contain a counter incremented periodically in accordance with certain clock signal and coupled with one or more comparator registers that cause interrupts to occur for specific values of the counter. Each of them is associated with a data structure organizing kernel timers in such a way that the interrupt handler can easily determine which timer will expire next and reprogram the clock event device to generate the next interrupt at that time[26]. System suspend unfortunately disturbs these operations, because it causes all of the clock event devices to be shut down, so they need to be programmed from scratch during the subsequent resume. Of course, to program them during the resume the kernel will use the new values of `xtime` and `wall_to_monotonic` that contain the rounding error resulting from the reinitialization of clock sources. These new values of `xtime` and `wall_to_monotonic` will also be compared with the timer expiration times when deciding which timers have already expired and their functions should be run. In consequence, all of the timers that had been added and didn't expire before the suspend are going to expire slightly earlier or slightly later than they would have expired if the suspend-resume cycle hadn't occurred. This means that the timing of some events in the system will be different from their analogous timing without the suspend-resume cycle, although the relative time intervals between events occuring after the resume will remain approximately the same.

Needless to say, a system that suspends and wakes up very often can introduce some confusion in a networking environment, because its idea about what the current time is will change with every suspend-resume cycle[27]. Moreover, NTP is not guaranteed to help in that case, because it also needs some time to resynchronize the kernel's monotonic clock with external time sources.

As stated above, on Linux-based systems this problem is connected with every form of full system suspend, not only with the opportunistic variant of it used by Android. However, if system suspend is initiated by user space, the kernel may assume that user space is ready for it and somehow prepared to cope with the consequences. For example, user space may want to take all of the network interfaces down before asking the kernel to suspend the system or to carry out other similar preparations. It also may want to use `settimeofday()` to set the kernel's monotonic clock using a time value taken from and NTP server right after the subsequent system resume. On the other hand, if system suspend is started by the kernel in an

[19] Defined in `kernel/time/timekeeping.c`.

[20] The total time spent in sleep states is accumulated in the `total_sleep_time` variable.

[21] On OMAP, though, it is much more accurate, because that platform provides a persistent clock ticking 32768 times per second.

[22] The $2\,s$ will also be added to `total_sleep_time`.

[23] On OMAP the rounding error is much smaller due to the higher resolution of the persistent clock, but still, if the frequency of the clock source used normally by the kernel is $10\,MHz$ or more, it may be several orders of magnitude.

[24] Roughly, it returns is the sum of `xtime` and `wall_to_monotonic` adjusted by the number of nanoseconds elapsed since the last modification of these variables, according to the clock source currently in use.

[25] On modern x86 PC machines they usually are Local Advanced Programmable Interrupt Controller (LAPIC) timers.

[26] The clock event device may also need to be reprogrammed when a new timer is added, if the new timer is to expire earlier than all the existing ones.

[27] Of course, it will also disappear from the network and reappear in it with every suspend-resume cycle and that may cause its own problems to happen.

opportunistic fashion, user space doesn't really have a chance to do anything like this.

4 Idle-Based System Power Management

In the Linux kernel there is a well implemented and widely used framework allowing CPUs to be put into low-power states when they have no code to execute, called *cpuidle* [19]. Although it is designed with CPU power management in mind, there seem to be natural ways to extend it so that it also covers power management of I/O devices. Moreover, there are good reasons to do that.

First, the fact that CPUs don't have any code to execute often indicates that there are no data to process for I/O devices. That need not be the case, though, because CPUs may be idle while data are being transferred between I/O devices and memory via DMA. Moreover, one CPU (or CPU core) may be idle while other CPUs (or CPU cores) in the system are executing instructions. Nevertheless, it might make sense to extend the *cpuidle* framework so that it could, for example, schedule the execution of the I/O runtime PM framework's `pm_request_idle()` function for certain set of I/O devices before putting the last CPU into a low-power state.

Second, there are integrated circuits, referred to as systems-on-a-chip (SoC), that consist of a CPU and a number of I/O devices depending on each other in various ways. In particular it is possible that the I/O parts of the chip have to be powered down before putting the CPU into a particular low-power state [20]. The *cpuidle* drivers for CPUs included in the chips designed in such a way have to take this limitation into account and there already are implementations doing that in the kernel tree, mostly in the ARM architecture subtree. Technically, however, this involves putting devices into low-power states from the context of the *cpuidle* framework when the CPU doesn't have any code to execute, so it really is not that much different from the case discussed in the previous paragraph.

It turns out that on some SoCs the state of the hardware after putting all of the I/O devices on the chip and the CPU into deep low-power states is very similar to the state of it after carrying out full system suspend [12]. Thus, seemingly, using the *cpuidle* framework to put all of the system's hardware components into deep low-power states one should be able to get substantial energy savings, comparable to the energy savings achievable with the help of the Android's kernel-based opportunistic suspend. Unfortunately, though, this really is not the case in general, since while the *cpuidle* framework may be able to put the system into the state in which the total power drawn by it is close to the minimum, referred to as the *pseudo sleep state* in what follows, it may not be able to do that sufficiently often.

In fact, although the ability to put the system into a state in which it draws (almost) minimum power is necessary for maximizing energy savings, it is not sufficient for this purpose. The amount of time spent in that state matters as well, because energy used by the system over a given time interval is the integral of the function representing power drawn by it over that interval [8]. For a given state of the system the contribution to the integral is the product of power drawn by the system in that state and the amount of time spent in it. For example, if the system has only two different states, A and B, and the power drawn by it in these states is denoted by P_A and P_B, respectively, then the energy used by it over a time interval of length T is given by

$$E = P_A T_A + P_B (T - T_A) \,,$$

where $T_A \leq T$ is the time spent by the system in state A. Thus the more time the system spends in the state in which it draws minimum power, the smaller the value of the integral and the less energy is used. Of course, if the system stayed in the pseudo sleep state all the time, it would use (almost) the minimum amount of energy, but then it couldn't do any useful work. Whenever useful work is done, the system draws more power than it would spending the same amount of time in the pseudo sleep state. Thus, in theory, to minimize the usage of energy, the system should go into the pseudo sleep state as soon as reasonably possible after the work has been completed and it should stay in that state for the maximum possible time. However, this turns out to be difficult to achieve in practice.

Every operation initiated by the *cpuidle* framework can only be carried out if one of the CPUs in the system becomes idle. Moreover, the system can only be put into the pseudo sleep state if all CPUs are idle and presumably all of them but one have been put into deep low-power states already. Hence, every application that causes one of the CPUs to execute instructions continuously (e.g. polling a file descriptor in a tight loop) is going to prevent the system from being put into the pseudo sleep state. At the same time, the Android's opportunistic suspend will trigger if there are no active wakelocks in the system, which very well may happen when some applications are running. On Android only the applications allowed to use wakelocks can prevent the system from being suspended and so long as they behave nicely, the system may be suspended relatively often. On the other hand, if the *cpuidle* framework is used to put the system into the pseudo sleep state, every single application that doesn't behave nicely affects the system's ability to save energy in the same way as a

"rogue" Android application using wakelocks. For this reason, the approach based on putting the system into the pseudo sleep state from the *cpuidle* framework's context can only be effective if *all* applications running on it are designed in accordance with specific set of rules, so that they sleep as often as reasonably possible and give the system a chance to minimize its total power draw relatively often. Of course, if the user can install arbitrary applications in the system, this is rather difficult to make happen.

In an attempt to address this issue one can divide all applications installed in the system into two sets, the ones that are regarded as "important", so they may be allowed to prevent the system from being put into the pseudo sleep state, and the ones that are regarded as "bulk", so they may be preemptively stopped when the "important" ones are sleeping to give the *cpuidle* framework a chance to do its job. Still, the fact that all of the "important" applications are sleeping alone need not imply that the "bulk" ones should be stopped. Namely, to stop the "bulk" applications one needs to know the reason why the "important" ones are sleeping, because they simply may be waiting for the "bulk" applications to do something. For instance, consider an e-mail client, regarded as "important", that starts an image viewer, regarded as "bulk", to display a graphics attachment. Usually, in that situation the e-mail client will go to sleep as soon as the image viewer is started, but that doesn't mean the image viewer should be stopped at the same time!

Regardless of the way the stopping of the "bulk" applications is implemented[28] there always will be some uncertainty about the intentions of the "important" applications, unless there is a mechanism allowing them to specify whether or not it is safe to stop the "bulk" ones. That mechanism, however, would have to be very similar to the Android wakelocks, although it might be confined to user space. Suppose, for example, that there is a power manager in user space deciding when to forcibly stop all of the "bulk" applications. For this purpose it will need to use IPC to allow the "important" applications to let it know whether or not they are waiting for the "bulk" ones. Thus it seems that there should be an IPC mechanism allowing an "important" process to send an "I'm waiting for process X" message to the power manager. It also should be possible to send an "I'm not waiting for any processes any more" message to the power manager from an "important" process. The power manager, in turn, will receive these messages and keep track of the number of "bulk" processes that are being waited for at any given time. If that number drops down to zero *and* all of the "important" processes are sleeping, the power manager will forcibly stop all of the "bulk" ones to let the *cpuidle* framework do its job. That, in theory, should help reduce the influence of the "bulk" applications on the system's ability to save energy, but at the cost of introducing the power manager and implementing the necessary IPC in the "important" applications.

Of course, this approach is susceptible to the problem of deciding which applications should be regarded as "important" that affects Android as noted in Section 3. Moreover, having introduced a user space power manager, one can use it to implement automatic suspend initiated from user space mentioned in Section 3, which is not very much different from the approach discussed in the previous paragraph. Still, if full system suspend is used to put the system into a sleep state, full system resume has to be used to put it back into the working state, which sometimes is problematic. For instance, suppose that user space has to check the battery status every 10 minutes. Then, if any form of system suspend is used to put the system into a sleep state, system resume has to be used every 10 minutes to bring the entire system, including all processes and I/O devices, back into the working state just in order to check the battery status[29]. Needless to say, this can lead to excessive energy usage during the time when the battery status is being checked (e.g. the "bulk" processes can cause the CPUs to execute instructions in that time intervals). In turn, if the approach described in the previous paragraph is used, only the "important" process checking the battery status has to be woken up and the parts of hardware it doesn't rely on may remain in deep low-power states. Therefore this approach seems to be worth investigating, although the system's ability to go into the pseudo sleep state relatively frequently may also be affected by power management Quality of Service (PM QoS) requests.

PM QoS may be used by kernel subsystems and user space processes to specify their expectations about certain aspects of the system's behavior related to power management [23]. Every expectation is expressed by a PM QoS request belonging to one of several PM QoS classes[30] that in turn is internally represented by the kernel in the form of a `struct pm_qos_request_list` object. For each PM QoS class the kernel maintains a priority-sorted list of requests, where the meaning of the priority depends on the class. In particular, for the `CPU_DMA_LATENCY` class, the priority is interpreted as the maximum expected latency of the transition of a CPU from a low-power state to the state in which it can execute code, in microseconds. Adding a PM QoS request one has to specify the priority of it, which for a `CPU_DMA_LATENCY` class request is the maximum

[28] There was a proposal to use the *cgroup freezer* feature to freeze them [21], but that is basically equivalent to sending `SIGSTOP` to all of them.

[29] This actually happens on Nexus One phones [22].

[30] At the time of this writing there are three PM QoS classes, the `CPU_DMA_LATENCY` class, the `NETWORK_LATENCY` class, and the `NETWORK_THROUGHPUT` class.

number of microseconds one can wait for a CPU to change states. At any given instant of time the only `CPU_DMA_LATENCY` class request that matters is the one with the lowest priority (i.e. the smallest acceptable CPU transition latency), because its priority is taken by the *cpuidle* governors into account when deciding which low-power state to put a CPU into.

Every CPU low-power state available to the *cpuidle* framework is associated with a specific latency of the CPU transition from that state to the functional state. If that latency is greater than the lowest `CPU_DMA_LATENCY` class PM QoS request priority, the CPU will not be put into the given state. Now, it seems reasonable to expect that the CPU transition latency associated with the system's pseudo sleep state will always be relatively high, so the majority of `CPU_DMA_LATENCY` class PM QoS requests will effectively prevent that state from being taken into consideration by *cpuidle* governors. In consequence, the users of the `CPU_DMA_LATENCY` class PM QoS requests can practically disable the system's ability to save substantial amounts of energy. In particular, this can be done by any `root`-owned user space processes, by opening the `/dev/cpu_dma_latency` special device file and writing a sufficiently low number to it. Of course, `root`-owned processes have much more potential to do harm than that, but if the *cpuidle* framework is going to be used for system-wide power management, one has to be aware of this potential difficulty related to PM QoS. Moreover, this is not the last possible problem with the *cpuidle*-based system power management.

In general, there may be one more difference between the pseudo sleep state that the system goes into as a result of the *cpuidle* framework's action and the "regular" sleep states entered by carrying out full system suspend. That potential difference is related to wakeup signals. More precisely, it is related to the fact that the number of devices configured to signal wakeup when the system is in the pseudo sleep state may be greater than the number of devices that can signal wakeup when the system is in one of the "real" sleep states. Namely, for the *cpuidle* framework making the system go into the pseudo sleep state it is most convenient to use runtime PM facilities, such as the I/O runtime PM framework, to put I/O devices into low-power states. In turn, these facilities, including the subsystem and driver callbacks used by the I/O runtime PM framework, generally work in such a way that all devices capable of signaling wakeup are configured to do so. On the other hand, only a limited set of specifically selected devices can wake up the system from "real" sleep states, which follows from both the software configuration and the capabilities of the available hardware. For example, some PCI devices are capable of signaling wakeup even if all the parts of the device not strictly necessary for that are entirely powered off (i.e. the device is in the "power off" state from the software point of view[31]) and they may be provided with the minimum amount of power needed to generate a wakeup signal while the system is in a sleep state[32]. Other PCI devices may not be capable of doing that, even though they generally can signal wakeup from "software accessible" low-power states $D1$–$D3$. These devices will generally be configured to signal wakeup before the system is put into the pseudo sleep state, but they will never be able to wake it up from a sleep state in which power is removed from the bus segment they are on.

Clearly, if wakeup signals are generated at approximately constant rate, the probability of waking up the system from an energy-saving state over a time unit is proportional to the number of devices capable of doing that, referred to as *wakeup sources* from now on. In turn, for a given probability of putting the system into an energy-saving state over a time unit, the amount of time spent in that state during a given time interval depends on the probability of waking up the system over a time unit. For example, if the system goes into the energy-saving state, on the average, once per second, while it is woken up every $2\,s$, it will spend from about $6\,s$ to about $7\,s$ in that state during a $10\,s$ interval. However, if it is woken up, on the average, every $5\,s$, it will spend between $8\,s$ and $9\,s$ in the energy-saving state during the same time interval. Thus, since the number of wakeup sources for a system in the pseudo sleep state is generally greater than the number of wakeup sources for the same system in a "real" sleep state, the amount of time spent by it, on the average, in the "real" sleep state generally will be greater, under the assumption that in both cases the probability of putting the system into the energy-saving state over a time unit is roughly the same[33]. Moreover, if the wakeup sources for the system in the pseudo sleep state include the clock event devices discussed in Section 3, the system will be woken up from that state every time a kernel timer expires, which may happen quite often on a typical Linux-based system.

In general, to mitigate this issue it would be necessary to turn off as many wakeup sources as possible while putting the system into the pseudo sleep state. That, in turn, might require the *cpuidle* framework to carry out some intrusive low-level operations beyond the "normal" I/O runtime PM. For PCI devices it would have to use special suspend callbacks turning off the devices' capability to wake up the system, unless they

[31] The official PCI documentation says that these devices can generate Power Management Events (PMEs) from the $D3_{cold}$ state.

[32] This minimum wakeup power supply is referred to as *auxiliary power* in the PCI documentation. In fact, usually, devices that can use auxiliary power can also signal wakeup after the whole system has been powered down, so long as enough power is provided to the motherboard by the power supply.

[33] That, of course, depends on how often the system can go into the given state, which has already been discussed earlier.

are supposed to wake it up from "real" sleep states. In consequence, leaving the pseudo sleep state would require the devices to be powered up, at least to be reprogrammed so that they could signal wakeup again. Similarly, if clock event devices, discussed in Section 3, were shut down when the system was going to the pseudo sleep state, they would need to be reprogrammed while leaving that state. All of these operations resemble the operations carried out during system suspend and resume very closely, except that user space is frozen when the system is suspending and resuming, which is not the case during transitions initiated by *cpuidle*.

While it may be easier to turn off wakeup sources from the *cpuidle* context on SoCs, the observation that runtime PM tends to prepare devices to signal wakeup, which generally is not desirable when the system is going to be put into the pseudo sleep state, remains valid for that platforms. Thus it seems that in order to make the system stay longer in the pseudo sleep state one would have to create a special framework for powering off devices that would be similar to the existing system suspend framework, except that it would need to satisfy additional requirements (i.e. it would have to be able to cope with occasional interactions with user space in case it becomes active in the middle of a transition). Moreover, if this difficulty is overcome and the system can stay in the pseudo sleep state until it is woken up by the user pressing a power button or by the RTC wake alarm discussed in Section 3, there still will be a problem with the values of `xtime` and `wall_to_monotonic`. Namely, if the time spent in the pseudo sleep state is simply added to `xtime` during wakeup without modifying `wall_to_monotonic`, it may be necessary to run a significant number of kernel timer functions simultaneously, because their timers have expired during that time. On the other hand, if the time spent in the pseudo sleep state is also subtracted from `wall_to_monotonic`, just like during "regular" system resume, the timers will expire later than they would have expired if the system had not been in that state, which still may lead to some undesirable effects[34]. Naturally, system suspend also is affected by this issue, but it would only make sense to work on increasing the average time spent in the pseudo sleep state if that allowed one to *avoid* issues associated with using the "real" sleep states. If these issues cannot be avoided, then it most likely is better to simply use system suspend instead.

To summarize, extending the *cpuidle* framework so that it can power manage I/O devices as well as CPUs appears to be a good idea in general and may be necessary for some specific types of hardware (i.e. SoCs in which the CPU is tightly coupled with I/O device in terms of power management). With the help of the mechanism causing "bulk" applications to be forcibly stopped in some situations discussed above, implemented entirely in user space, it may be possible to allow the system to enter a state in which its total power draw is very close to the minimum relatively often. However, attempting to introduce mechanisms that would allow the system to stay in that state as long as in a sleep state (entered as a result of carrying out full system suspend) doesn't look like a promising way to go. It may be worth doing on some types of hardware, but the prospective complications associated with it are not encouraging. On the other hand, though, if the usage of timers in the kernel is optimized, which very likely is going to happen for other reasons [24], the idle-based system power management will automatically become more attractive.

5 Wakeup Events Framework and Opportunistic Suspend

The Android's wakelocks framework discussed in Section 2 attempts to address multiple problems with one unified approach, akin to the famous Swiss Army knife. First, it allows the kernel to avoid races between the suspend process and wakeup events (i.e. wakelocks are activated whenever wakeup events occur, which prevents the kernel from suspending the system or causes it to abort suspend in progress). Second, it provides a straightforward criterion for the kernel to decide when opportunistic suspend should be started (i.e. the kernel attempts to suspend the system whenever there are no active wakelocks). Moreover, it helps to avoid the problem with "bulk" applications that may cause the system to use too much energy, described in Section 4 (i.e. "bulk" applications are not allowed to use wakelocks and therefore they cannot prevent the kernel from suspending the system). In addition to that it makes the kernel collect statistics related to wakeup events. Namely, each wakelock object contains fields in which statistical information is recorded, like the number of times the wakelock has been activated, the total time it has been active, or the maximum time it has been active continuously, whenever it is activated or deactivated[35].

Out of the four problems addressed by the wakelocks framework mentioned above two matter only if

[34]Consider a kernel subsystem that uses a timer to implement a timeout of 5 *s*. After 1 *s* the system goes into the pseudo sleep state and spends 2 *s* in it, but the time spent in that state is subtracted from the monotonic clock value. Then, the timer will expire 4 *s* after the wakeup from the pseudo sleep state, which in fact is 7 *s* after it was started, instead of the original 5 *s*.

[35]On Android this feature is also used to annotate code. Its implementation of wakelocks requires their users to give them names that are associated with the reported statistics. Then, the names given to wakelocks can be used to easily trace the pieces of code that use them. That applies to the wakelocks manipulated by user space too.

kernel-based opportunistic suspend is used. Still, the races between the suspend process and wakeup events affect system suspend started from user space too, so there should be a mechanism for avoiding them in that case. Collecting statistics related to wakeup events is also generally useful, so it would be good to have an infrastructure for that either. In principle it might be possible to use the wakelocks framework as is for this purpose, but that wouldn't be very convenient.

The wakelocks framework is really concerned with whether or not the processing of all wakeup events is complete and, consequently, whether or not it is possible to suspend the system at a given instant of time. That's why it allows user space processes that may take part in the processing of wakeup events to use wakelocks in the first place. However, this means that within the wakelocks framework the races between wakeup events and the suspend process can only be avoided if the relevant user space processes are modified to use wakelocks. While that may be good for Android, whose user space already is designed with using wakelocks in mind, at least to some extent, it is not very practical for other Linux-based systems with user space which is not aware of the wakelocks interface. Still, the possible races between the suspend process and wakeup events affect those systems as well and there should be a way to avoid them without redesigning user space completely.

Suppose you want to initiate suspend from user space, by writing `mem` to `/sys/power/state`, but at the same time you want to avoid races with wakeup events. In other words, if one of the devices configured to wake up the system from the memory sleep state signals wakeup while the system is suspending or even a little before you write to `/sys/power/state`, the suspend operation should not succeed. There's a question, however, how much time before writing to `/sys/power/state` is really interesting. Are wakeup events signaled 10 minutes before writing to `/sys/power/state` so important that they should cause the suspend to fail? Is $10\,s$ a more reasonable time distance? The kernel doesn't really know that in advance and it has to be provided with that information by user space. That may be accomplished with the help of an interface allowing user space to tell the kernel: "If any wakeup events are signaled from now on, the subsequent attempt to suspend should fail."

That leads to the question how to detect whether or not any wakeup events were signaled in a given time interval. While in principle there are multiple possible ways to achieve that, probably the simplest of them is to have a running counter of signaled wakeup events and to compare its values at the beginning and at the end of the time interval of interest: If they are different, at least one wakeup event has occurred in the meantime. Hence, if there is an interface for kernel subsystems to increment the counter whenever a wakeup event is signaled and another interface for user space to tell the kernel what value of the counter to take as the initial one for the given attempt to suspend, it should be possible to avoid races between wakeup events and the suspend process, so long as all of the wakeup events can be regarded as instantaneous.

In reality, though, there are wakeup events that produce data, like a key press on a keyboard enabled to wake up the system, and these data need to be processed or at least passed to user space before the system may be permitted to suspend. It may be thought that these events are not instantaneous, but take some time to receive, starting at the moment when wakeup is signaled and ending at the moment when its data have been delivered to user space. For wakeup events of this type the counter introduced in the previous paragraph should only be incremented when the processing of the event by the kernel has ended, but at the same time the system should not be suspended after the event has been signaled. This complication may be taken into account by using a counter of wakeup events whose processing by the kernel hasn't ended yet in addition to the counter introduced above. This new counter will be incremented whenever there is a wakeup event producing data to be processed by the kernel. In turn, it will be decremented after the processing of the event's data by the kernel has ended (e.g. the data have been passed to user space) and that will cause the main counter of signaled wakeup events to be incremented. Accordingly, an attempt to suspend the system will fail if the counter of wakeup events whose data are being processed by the kernel is not equal to zero. Of course, for that to work, there ought to be an interface for kernel subsystems to mark the beginning and ending of a "continuous" wakeup event and it is necessary to modify the core power management (PM core) code to actually use both counters.

The above considerations led to the kernel patch that introduced the wakeup events framework shipped in the 2.6.36 kernel [25]. Specifically, it introduced the running counter of signaled wakeup events, `event_count`, and the counter of wakeup events whose data are being processed by the kernel, `events_in_progress`[36]. Two interfaces have been added to allow kernel subsystems to modify these counters in a consistent way. First, the `pm_stay_awake()` function increments `events_in_progress` and the complementary function `pm_relax()` decrements it and increments `event_count` at the same time. Additionally, `pm_stay_awake()` takes a `struct device` pointer argument in order to update the new `power.wakeup_count` field in the device object

[36]Defined in `drivers/base/power/wakeup.c`.

pointed to by it, which is exported via *sysfs* (read-only) as `/sys/devices/.../power/wakeup_count`[37]. Second, the function `pm_wakeup_event()` increments `events_in_progress` and sets up a timer to decrement it and increment `event_count` in the future. Its first argument is a `struct device` pointer used for updating the device's `power.wakeup_count` field and its second argument is the number of milliseconds to wait before decrementing `events_in_progress`.

The current value of `event_count` can be read from the new *sysfs* file `/sys/power/wakeup_count`. In turn, writing to it causes the current value of `event_count` to be stored in the auxiliary variable `saved_count`. However, the write operation will only succeed if the written number is already equal to `event_count`. If that happens, another auxiliary variable `events_check_enabled` is set, which tells the PM core to use `event_count` and `events_in_progress` during the subsequent suspend[38]. Namely, the PM core's suspend code calls the function `pm_check_wakeup_events()` returning `true` if either `events_check_enabled` is not set, or `events_in_progress` is equal to zero *and* `event_count` is equal to `saved_count`. Otherwise, `false` is returned which means that the system should not be suspended.

This relatively simple mechanism allows the PM core to react to wakeup events signaled during system suspend if it is asked to do so by user space and if the kernel subsystems that detect wakeup events use either `pm_stay_awake()` or `pm_wakeup_event()` to let the PM core know about them[39]. Arguably, it may be used to implement opportunistic suspend based on a power manager in user space, which is discussed in more detail below. Still, its support for collecting statistics related to wakeup events is not comparable to the one provided by the wakelocks framework by any means. Moreover, it assumes that wakeup events will always be associated with devices, or at least with entities represented by device objects, which need not be the case in all situations. The need to address these shortcomings led to a kernel patch introducing wakeup source objects and adding some flexibility to the existing framework [26].

The new patch[40] leaves the interfaces described in a few previous paragraphs basically untouched, with one exception. Namely, it makes the `pm_relax()` function take a `struct device` pointer argument and allows it to be used for canceling timers set up by `pm_wakeup_event()`. More precisely, if `pm_wakeup_event()` is used to report a wakeup event for certain device object, there's no need to wait until the timer set up by it expires. If it is known in advance that the kernel processing of the event has been completed, `pm_relax()` may be called directly for the given device and the timer set up by `pm_wakeup_event()` will be canceled. Of course, `events_in_progress` and `event_count` will be updated as appropriate in that case.

Under the hood, however, quite a lot is being changed. Most importantly, the new patch introduces objects of type `struct wakeup_source` to represent entities that can generate wakeup events. Each wakeup source object contains, among other things, several fields holding statistical information, a timer used by `pm_wakeup_event()`, a name and the flag called `active` that is set whenever wakeup events associated with this object are being processed. Wakeup source objects are created automatically for devices that are enabled to signal wakeup. Specifically, when `device_set_wakeup_enable()` is called for a given device with the second argument equaling `true`, a wakeup source object is created and added to the system-wide list of wakeup source objects. Its address is stored in the device's `power.wakeup_source` field and used internally by `pm_wakeup_event()`, `pm_stay_awake()` and `pm_relax()` called for that device. It is destroyed when `device_set_wakeup_enable()` is called for its device with the second argument equal to `false`. The statistical information stored in a device's wakeup source object is accessible by user space via *sysfs* as the contents of several new files in the `/sys/devices/.../power/` directory corresponding to the given device[41].

Although the highest-level interfaces are still designed to report wakeup events relative to devices, which is particularly convenient to device drivers and subsystems that generally deal with device objects, like the PCI bus type, the new framework makes it possible to use wakeup source objects directly for reporting wakeup events. "Standalone" wakeup source objects are created using `wakeup_source_create()` and added to the

[37] This file may be used by user space to check the number of wakeup events associated with the given device.

[38] An inquiring reader may wonder why to complicate things this way. It seems that it might be sufficient to simply read from `/sys/power/wakeup_count` to "snapshot" the current value of `event_count` and then try to suspend, so the required write operation appears to be redundant. However, it sometimes is necessary to suspend *without* taking wakeup events signaled during suspend to account. Namely, suppose that user space has already written to `/sys/power/wakeup_count` and *then* it wants to change its mind and ignore wakeup events occurring during the subsequent suspend. In that case it only needs to read from `/sys/power/wakeup_count`, which resets `events_check_enabled`, and start the suspend afterwards. Another reason why it is done this way is that the hibernate code can check for wakeup events too and it may be run from the `ioctl()` interface of `/dev/snapshot`, in which case it's better to avoid opening `/dev/snapshot` if some wakeup events are known to have been signaled beforehand instead of opening it and calling `ioctl()` on it just to learn that it failed.

[39] The PCI subsystem is the only one doing that at the moment.

[40] Currently scheduled for inclusion into the 2.6.37 kernel after receiving favorable reviews.

[41] The `wakeup_count` field corresponding to one of these files has been moved from the `power.wakeup_count` member of `struct device` to the `struct wakeup_source` object pointed to by the `power.wakeup_source` member of `struct device`.

kernel's global list of wakeup sources by calling `wakeup_source_add()`. The `wakeup_source_register()` function can be used to carry out these two operations in one shot. Afterwards one can use three new interfaces, `__pm_wakeup_event()`, `__pm_stay_awake()` and `__pm_relax()`, completely analogous to the functions with corresponding names described above, except that they take a pointer to `struct wakeup_source` instead of a pointer to a device object. When a wakeup source object is not necessary any more, it may be removed from the global list of wakeup sources by calling `wakeup_source_remove()` and then deleted with the help of `wakeup_source_destroy()`. Alternatively, these two operations can be carried out together by calling `wakeup_source_unregister()`. Thus reported wakeup events need not be associated with any device objects any more. Moreover, at the kernel level, wakeup source objects may be used to replace Android's wakelocks on a one-for-one basis, because the interfaces for manipulating them are completely analogous to the ones introduced by the wakelocks framework. This ought to make it easier to port device drivers from Android to the mainline kernel code base without merging the entire wakelocks, or suspend blockers, framework[42].

The infrastructure described earlier in this section has been introduced to make it possible to avoid races between wakeup events and a typical suspend process started by user space. It is not designed with opportunistic or even less aggressive automatic suspend in mind, but in theory it may be used for implementing such power management techniques. Of course, since it doesn't make the kernel decide when to suspend the system, a user space power manager is necessary for this purpose. In general, it will talk to the kernel through the *sysfs* files in `/sys/power/` and will exchange information with other user space processes using some kind of IPC. In principle it may use arbitrary heuristics in making its decisions, but one special case seems to be particularly worth considering. Namely, it is instructive to check if the Android's opportunistic suspend can be emulated with the help of it.

First, all wakelocks in the Android kernel can be replaced with wakeup source objects thanks to the correspondence between them mentioned above. Then, if the `/sys/power/wakeup_count` interface is used correctly, the resulting kernel will be able to abort suspend in progress in reaction to wakeup events in the same circumstances in which the original Android kernel would do that. Yet, user space cannot access wakeup source objects, so the part of the wakelocks framework allowing user space to manipulate them has to be replaced with a different mechanics implemented entirely in user space, involving a power manager process and a suitable IPC interface for the processes that would use wakelocks on Android.

The IPC interface in question may be implemented using three components, a shared memory location containing a counter variable referred to as the "suspend counter" in what follows, a mutex, and a conditional variable associated with that mutex [27]. Then, a process wanting to prevent the system from suspending will acquire the mutex, increment the suspend counter and release the mutex. In turn, a process wanting to permit the system to suspend will acquire the mutex and decrement the suspend counter. If the suspend counter happens to be equal to zero at that point, the processes waiting on the conditional variable will be unblocked. The mutex will be released afterwards. These two procedures are referred to, respectively, as the "block suspend" and "unblock suspend" operations below. They ensure that only one process at a time can access the suspend counter and possibly unblock processes waiting on the conditional variable.

With the above IPC interface in place the power manager process can perform the following steps in a loop:

1. Read from `/sys/power/wakeup_count`. This will block until the kernel's `events_in_progress` variable is equal to zero.
2. Acquire the mutex. This prevents the power manager from accessing the suspend counter while any block suspend or unblock suspend operations are in progress. It also prevents any suspend block and unblock operations from updating the suspend counter when the power manager is reading its value.
3. Check if the suspend counter is equal to zero. If that's not the case, block on the conditional variable (that will automatically release the mutex). When unblocked, go to step 2.
4. Release the mutex.
5. Write the value read from `/sys/power/wakeup_count` in step 1 back to this file. If the write fails, go to step 1.

[42]One of the main problems stemming from differences between the mainline kernel and the Android's one is that device drivers developed for Android are generally required to use wakelocks. In consequence, they cannot be submitted for inclusion into the mainline kernel in the form in which they are actually *used*, because it doesn't include the wakelocks framework. Although one can argue that it should be possible to remove all references to wakelocks from them and submit them in that form, that would require their developers to maintain two different versions of every driver at the same time, the Android one actually used and the mainline one without wakelocks. Keeping them in sync would be troublesome to put it lightly and it is no wonder that no one is willing to do that. On the other hand, if the mainline kernel provides interfaces analogous to the wakelocks' ones, the whole "porting" will boil down to renaming a few functions and data types.

6. Start suspend or hibernation (e.g. by writing the appropriate string to `/sys/power/state`) and go to step 1 when it returns.

This way it cannot deadlock with the processes carrying out the block suspend and unblock suspend operations, because it never goes to sleep with the mutex held and the other processes don't do that either. Moreover, if they do the right things, the system will not be suspended at a wrong time. Namely, if a wakeup event is reported at the kernel level, the application receiving it from the kernel should carry out a suspend block operation before the kernel thread processing the event calls `__pm_relax()` for its wakeup source. Then, if `__pm_stay_awake()` was called for that wakeup source after the power manager had read from `/sys/power/wakeup_count` in step 1, either the write in step 5, or the attempt to suspend in step 6 will fail, because the kernel's variable `event_count` has changed in the meantime. In turn, if `__pm_stay_awake()` had been called before the power manager attempted to read from `/sys/power/wakeup_count`, the read operation can only complete after `__pm_relax()` is called for the given event's wakeup source. In that case the power manager will see that the suspend was blocked in step 3 and it will go to sleep. It will be woken up when the suspend counter goes back to zero and the write operation in step 5 will allow it to verify if more wakeup events were reported at the kernel level[43].

Of course, the above design will cause the system to be suspended very aggressively. Although it is not entirely equivalent to the opportunistic suspend feature on Android, it appears to be close enough to yield the same level of energy savings. It doesn't include any facilities for recording statistical information related to the user space's usage of the suspend block and suspend unblock operations, but they may be added to it relatively easily. However, it also suffers from a number of problems affecting the Android's opportunistic suspend implementation. Some of them may be addressed by adding complexity to the power manager and the IPC interface between it and the processes permitted to block and unblock suspend. For example, the power manager can maintain a list of instants of time when system should be woken by the RTC wake alarm and can program the RTC as appropriate every time before suspending the system, as suggested in Section 3. Still, some other issues, like the appearance of inaccuracies of the kernel's monotonic clock resulting from every suspend-resume cycle, also discussed in Section 3, are not really avoidable. For this reason, it may be better to use system suspend less aggressively, but in combination with some techniques described in Section 4.

Overall, while the idea of suspending the system extremely aggressively may be controversial, it doesn't seem reasonable to entirely dismiss automatic suspending of it as a valid power management measure. Many different operating systems do that and they achieve good battery life with the help of it [5]. In the not so humble opinion of yours truly, there are no valid reasons why Linux-based systems shouldn't do that, especially if they are battery-powered. As far as desktop and similar (e.g. laptop or netbook) systems are concerned, it makes sense to configure them to suspend automatically in specific situations so long as system suspend is known to work reliably on the given configuration of hardware. The new interfaces and ideas presented above may be used to this end.

That said, automatic or even opportunistic suspend should never be the only way to suspend the system, because there are situations in which the user simply wants to suspend it *right away* regardless of whether or not there are any outstanding wakeup events to process. Two quite common examples of that come to mind. First, the user may be in a hurry and may be packing the device into a bag or a pocket without any intention to look into it any time soon. In such a situation a refusal to suspend the system because of some wakeup events appearing at a wrong time may be *very* annoying. Second, imagine that the system's battery is almost completely exhausted and the only way to protect it from a surprise powering off is to hibernate it as quickly as possible. In that case a refusal to do what the user is asking for may even lead to data loss, so it's better to discard all wakeup events until the system actually enters the sleep state. Thus it always is nice if the user can order the system to suspend or hibernate and *ignore* wakeup events signaled in the process. One has to wonder, though, if that's not too much to ask system designers for.

6 Conclusion

The Android's wakelocks framework, described in Section 2, was designed so that system suspend could be used as an aggressive energy-saving measure, essentially instead of runtime power management. Admittedly, at the time the first version of Android was released this opportunistic suspend approach was practically the only reliable way to achieve good battery life on a Linux-based system like the Google G1. The kernel's support for runtime power management was then totally focused on CPUs, I/O runtime PM features were

[43]This doesn't protect the power manager from racing with "wakeup events" originating in user space, but it's rather difficult to imagine a valid usage scenario in which that would really matter.

ad hoc if they existed at all for the hardware platform of interest and the available hardware could not be put into sufficiently deep low-power states with the help of the *cpuidle* framework alone. On the other hand, the system suspend framework was basically usable, so it was more convenient to choose as the basis for the new system's power management infrastructure, notwithstanding some problems mentioned in Section 3.

Opportunistic suspend still works rather nicely on Android today, which most probably is why its developers are not really interested in any alternatives. Yet, the mainline kernel has evolved since the first Android release and it is now offering much better support for runtime power management than ever before. Moreover, the hardware that newer versions of Android usually run on also is more sophisticated, so in many cases it may be powered down almost completely as a result of a transition initiated by *cpuidle*. While transitions of this kind need not occur sufficiently often in general, that situation may be improved by reducing the "bulk" applications' ability to keep the CPUs busy all the time, as described in Section 4. Unfortunately, though, the system may not be able to stay in the energy-saving state reached as a result of a *cpuidle*-initiated transition long enough, mostly because of kernel timers and interrupts generated whenever they expire. This problem is rather difficult to tackle. Seemingly, it may be mitigated by optimizing the usage of timers in the kernel rather than by adding complexity to the *cpuidle* framework's code paths. Anyway, things should get better over time in that area too.

By far the most controversial aspect of the Android's opportunistic suspend infrastructure is that it starts system suspend from kernel space, basically taking over an interface that was intended for use by (privileged) user space processes. Because of that it requires applications to interact with the kernel in a very unusual way and the interface provided by it for this purpose doesn't really match other interfaces between the kernel and user space. At the same time, though, it addresses the problem of possible races between the suspend process and wakeup events which is real and affects the mainline kernel's suspend and hibernation subsystem. It also provides means for collecting statistics related to wakeup events, which generally is a useful feature. Undoubtedly it would be good if the mainline kernel could do that too, but the existing suspend and hibernation interfaces also should be able to benefit from that without far-reaching modifications of user space. Hopefully the new wakeup events framework outlined in Section 5 will help to make that happen.

From the perspective of kernel subsystems outside of the PM core the wakeup events framework is analogous to the Android's wakelocks framework, but it allows one to report wakeup events relative to devices rather than relative to abstract entities such as wakelocks. Thus the parts of the kernel that already use device objects need not create and maintain additional data structures just for reporting wakeup events. Moreover, to take advantage of the user space interface introduced by the wakeup events framework it only is necessary to modify the part of user space that drives the suspend process. The other parts of user space need not be changed for this purpose in general, although it may be worthwhile to change them too, depending on the specific needs of the system. In particular, if the system needs to use an opportunistic suspend feature analogous to the Android's one, it may be implemented with the help of the wakeup events framework, a user space power manager and a special IPC interface, as described in Section 5. This means, however, that the most controversial parts of the Android's opportunistic suspend infrastructure are not really necessary and therefore they should not be included into the mainline kernel.

Furthermore, because of the similarities between the new mainline kernel's wakeup events framework and the Android's wakelocks framework, it should be possible to convert the vast majority of the Android device drivers using wakelocks to the mainline kernel code base. In consequence, it ought to be relatively easy to merge these drivers into the mainline kernel without the necessity to create special versions of them devoid of wakelocks, which from the mainline kernel developers' standpoint is a sufficient outcome. It remains to be seen whether Android developers decide to use the new wakeup events framework or they prefer to stick to the kernel-based opportunistic suspend and carry a patch against the mainline kernel for that. As far as the mainline kernel is concerned, what matters is the possibility to merge the Android device drivers without major modifications.

References

[1] Jonathan Corbet, *Blocking suspend blockers* (`http://lwn.net/Articles/388131/`).

[2] Rafael J. Wysocki, *Re: [PATCH 0/8] Suspend block api (version 8)* (`http://www.spinics.net/lists/linux-pm/msg19414.html`).

[3] Matthew Garrett, *Android/Linux Kernel: Lessons Learned* (`http://events.linuxfoundation.org/slides/2010/linuxcon2010_garrett.pdf`).

[4] Len Brown, Rafael J. Wysocki, *Suspend to RAM in Linux* (`http://www.linuxsymposium.org/archives/OLS/Reprints-2008/brown-reprint.pdf`).

[5] Len Brown, *Saving Energy with the intel_idle cpuidle Driver* (http://events.linuxfoundation.org/slides/2010/linuxcon2010_brown.pdf).

[6] Paul Fox, *OLPC's power manager* (http://dchs.spinics.net/lists/linux-pm/msg21483.html).

[7] Jonathan Corbet, *Another wakeup event mechanism* (http://lwn.net/Articles/393314/).

[8] Rafael J. Wysocki, *Runtime Power Management Framework for I/O Devices in the Linux Kernel* (http://events.linuxfoundation.org/slides/2010/linuxcon2010_wysocki.pdf).

[9] Brian Swetland, *Re: [PATCH 0/8] Suspend block api (version 6)* (http://lkml.org/lkml/2010/5/17/496).

[10] *PCI Local Bus Specification Rev. 3.0* (http://www.pcisig.com/members/downloads/specifications/conventional/PCI_LB3.0-2-6-04.pdf).

[11] *PCI Bus Power Management Interface Specification Rev. 1.2* (http://www.pcisig.com/members/downloads/specifications/conventional/pcipm1.2.pdf).

[12] Arve Hjønnevåg, *Re: [PATCH 0/8] Suspend block api (version 8)* (http://lkml.org/lkml/2010/6/1/19).

[13] Jonathan Corbet, *What comes after suspend blockers* (http://lwn.net/Articles/390369/).

[14] Alan Cox, *Re: [PATCH 0/8] Suspend block api (version 8)* (http://lkml.org/lkml/2010/5/27/228).

[15] *Information security* (http://en.wikipedia.org/wiki/Information_security).

[16] *Real-time clock alarm* (http://en.wikipedia.org/wiki/Real-time_clock_alarm).

[17] *High resolution timers and dynamic ticks design notes* (http://www.mjmwired.net/kernel/Documentation/timers/highres.txt).

[18] *Time Stamp Counter* (http://en.wikipedia.org/wiki/Time_Stamp_Counter).

[19] Jonathan Corbet, *The cpuidle subsystem* (http://lwn.net/Articles/384146/).

[20] Kevin Hilman, *RFC: mixing device idle and CPUidle or non-atomic idle notifiers* (http://kerneltrap.org/mailarchive/linux-kernel/2010/9/24/4624182).

[21] Paul E. McKenney, *Suspend blockers summary, Epilogue 1* (http://lwn.net/Articles/400771/).

[22] Arve Hjønnevåg, *Re: [PATCH 0/8] Suspend block api (version 8)* (http://www.mail-archive.com/linux-omap@vger.kernel.org/msg29710.html).

[23] Mark Gross, *Power Management Quality of Service (PM_QOS)* (http://elinux.org/images/f/f9/Elc2008_pm_qos_slides.pdf).

[24] Jonathan Corbet, *Solid-state storage devices and the block layer* (http://lwn.net/Articles/408428/).

[25] Rafael J. Wysocki, *PM: Avoid losing wakeup events during suspend* (http://lwn.net/Articles/392897/).

[26] Rafael J. Wysocki, *PM / Wakeup: Introduce wakeup source objects and event statistics (v2)* (http://lwn.net/Articles/405108/).

[27] Ulrich Drepper, *Futexes Are Tricky* (http://people.redhat.com/drepper/futex.pdf).