Boosting up Embedded Linux device: experience on Linux-based Smartphone

Kunhoon Baik
Samsung Electronics Co., Ltd.
knhoon.baik@samsung.com

Saena Kim
Samsung Electronics Co., Ltd.
saina.kim@samsung.com

Suchang Woo
Samsung Electronics Co., Ltd.
suchang.woo@samsung.com

Jinhee Choi
Samsung Electronics Co., Ltd.
jh106.choi@samsung.com

Abstract

Modern smartphones have extensive capabilities and connectivities, comparable to those of personal computers (PCs). As the number of smartphone features increases, smartphone boot time also increases, since all features must be initialized during the boot time. Many fast boot techniques have focused on optimizing the booting sequence. However, it is difficult to obtain quick boot time (under 5 seconds) using the fast boot techniques, and many parts of the software platform require additional optimization. An intuitive way to obtain instant boot times, while avoiding these issues, is to boot directly from hibernation. We apply hibernation-based techniques to a Linux-based smartphone, and thereby overcome two major obstacles: long loading times for snapshot image and maintenance costs related to hardware change.

We propose two mechanisms, based on hibernation, to obtain outstanding reductions in boot time. First, minimize the size of snapshot image via page reclamation, which reduces the load time of image. Snapshot is split into two major segments: *essential-snapshot-image* and *supplementary-snapshot-image*. The essential snapshot image is a minimally-sized image used to run the Linux kernel and idle screen, and the supplementary-snapshot-image contains the remained that could be restored on demand. Second, we add additional device information to the essential-snapshot-image, which is used when the the device is reactivated upon booting up. As a result, our mechanism omits some time-consuming jobs related to device re-initialization and software state recovery. In addition to quick boot times, our solution is low maintenance. That is, while the snapshot boot[3] is implemented in the bootloader, our solution utilizes the kernel infrastructure because it is implemented in the kernel. Therefore, there is little effort required, even when the target hardware is changed. We prototyped our quick boot solution using a S5PC110[17]-based smartphone. The results of our experiments indicate that we can obtain get dramatic gain in performance in a practical manner using this quick boot solution.

1 Introduction

Smartphones generally require long boot times. As the number of smartphone functions increases, the initialization times required for the corresponding software modules also increase. In addition, as smartphones are equipped with more and more peripheral devices such as sensors, cameras, Bluetooth and WiFi, these devices require their own initialization times, which further increases boot time.

To obtain instant boot times, "boot optimization" or "hibernation-based boot" techniques can be used. In the case of "boot optimization", each module must be optimized and the initialization flow must be modified after a profiling step. This can be difficult to accomplish if there are many software modules involved, or if the initialization process is complex. However, in the case of "hibernation-based boot" techniques, we can obtain instant boot times quite easily. In this paper, we apply hibernation-based fast boot techniques to a Linux-based smartphone.

There remain some barriers to applying hibernation-based boot techniques.

1. Mobile software platforms, such as Android[14], hold about 100MB of RAM capacity, but Flash

memory offers only poor I/O speed. If the read performance is 20MB/s, then snapshot image alone require loading time of about 5 seconds. Therefore, we cannot obtain instant boot times via the hibernation-based boot technique alone.

2. Because swsusp's the device reactivation flow in the standard Linux kernel was developed for generic purposes[1], it has some additional steps to reactivate devices. The snapshot boot technique eliminates these steps by restoring snapshot image in the bootloader, but also requires additional implementations in the bootloader.

3. If the same snapshot image is used every time the device boots up, information inconsistency problems will occur in the file system and database.

In this paper, we introduce new methods to obtain instant boot times by solving these issues. We focus on the following two methods. The first method optimizes the size of snapshot image to be less than 15MB without compression, to reduce snapshot image loading time. The second method improves the device reactivation flow to obtain similar performance to the snapshot boot technique, without tinkering with the bootloader. We also briefly discuss related issues such as information inconsistency problems.

This paper is organized as follows. In section 2, we summarize fast boot techniques already developed in Embedded Linux systems, and compare them with our approach. In section 3, we analyze smartphone boot times and investigate points where improvements can be made. In section 4, we introduce our approach, which optimizes snapshot image loading times with on-demand-paging and early device reactivation. Section 5 describes the experimental environment and provides experimental results. Finally, in sections 6 and 7, we suggest directions for future work and summarize the paper.

2 Related studies

Until recently, Linux development has focused on the desktop and server markets, in which boot time is not an important issue. However, boot time has become an important feature as more and more embedded systems are adopting Linux due to benefits such as low cost and the ability to be utilized across a variety of hardware platorms.

Boot time optimization techniques include profiling, reduction and optimizing techniques. These techniques were well summarized by the Bootup Time Working Group of the CE Linux Forum[5]. This section introduces some of these techniques, which can be used with our approach.

At the bootloader level, uncompressed kernel[6] or fast kernel decompression[7] techniques can be used. In the case of uncompressed kernel techniques, the kernel image loading time is longer but decompression time is not required. Fast kernel decompression improves kernel decompression performance using fast decompress mechanisms such as UCL[18]

At the kernel level, disable console[8], preset loops per jiffy(LPJ)[9] and deferred initcalls[10] techniques may be used. The disable console technique minimizes kernel printk messages during boot time to reduce serial console accessing time. The preset LPJ uses a constant delay value instead of the `calibrate_delay()` function that is commonly used for calibrating delay time in the kernel. The deferred initcalls technique forces some initcalls to run later if they do not need to be initialized early.

Hibernation-based techniques also reduce boot time. Hibernation is a feature used for power management in Linux. As a power saving mode, hibernation backs up the running state of the system into the disk space as a snapshot image, and powers down the system. When the power comes back up, the system is restored to the running state based on the snapshot image. Hibernation can be implemented by several techniques. Among these, the most common techniques are the swsusp technique that is included in the standard Linux kernel, and the TuxOnIce(suspend2)[11] technique that is provided as a patch of the kernel. The fundamentals are almost the same in these two techniques, but TuxOnIce offers more useful options compared to swsusp. However, the TuxOnIce patch requires many changes for the kernel, and therefore also incurs additional maintenance costs according to the kernel revision. The snapshot boot technique is a fast boot technique based on swsusp. In this technique, every time a device boots up, the snapshot

[1] swsusp (Software Suspend) is a suspend-to-disk implementation in the 2.6 series Linux kernel. It is the Linux equivalent of Windows hibernate functionality.

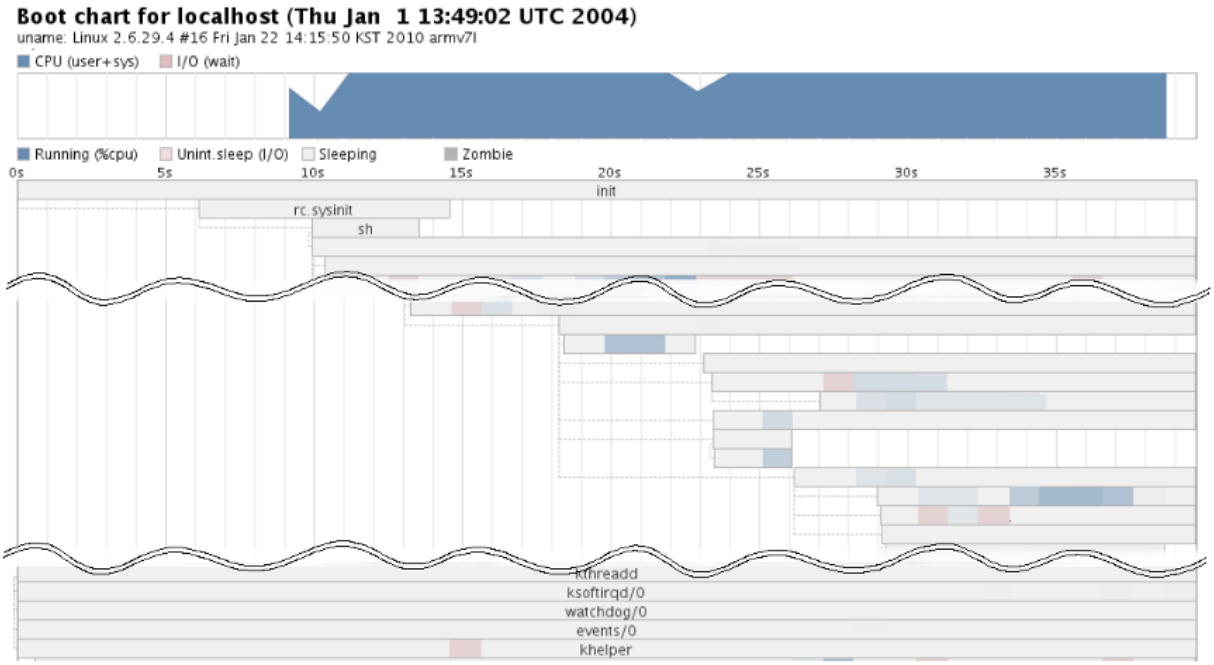

Figure 1: Bootchart – normal boot sequence of the smartphone used in this study

image is loaded in the bootloader instead of the original kernel image. Device initialization tasks are also performed at the bootloader level to improve the device reactivation flow in swsusp. However, most of the changes in the bootloader are heavily dependent on hardware, and for this reason, the associated maintenance costs are increased due to required changes of hardware design. This shortcoming makes the application of snapshot boot techniques less practical. Even if the snapshot boot technique is applied to a system, instant boot may not be achieved without further optimizing the size of the snapshot image. Recent smartphones require more memory space than older models, because of their extensive functionalities, and for this reason optimizing snapshot image must be considered a necessity. In a previous case study examining the use of the snapshot boot technique for digital TV systems[4] many parts of the software platform were modified to minimize the size of the snapshot image. However, such an approach increases maintenance costs due to the necessity of hardware and software platform revisions.

3 Smartphone Boot Time

Figure 1 is a the bootchart[12] of the smartphone model used in our experiments. More than 30 seconds of boot time are required to initialize the user area. This indicates that it will be difficult to reduce boot time to less than 5 seconds by optimizing the boot sequence. Even if we implement hibernation-based fast boot techniques, we cannot achieve 5 second boot times due to the barriers described in section 1.

To solve these problems, we must analyze each element of hibernation-based boot time. We calculate hibernation-based boot time(t_b) using the following formula.

$$t_b = t_p + t_l + \sum t_r$$

t_p includes the block device setup time for loading snapshot image and the cpu/clock/timer/power setup times for minimal operation. These constitute the necessary initialization events for booting from hibernation, and therefore the time required for these steps cannot be reduced. t_l is the time required to load the image from disk to the original memory location, and t_r is the time required to restore the cpu/device to the same state as when the snapshot image was made. These two factors can be optimized by improving the implementation of hibernation. t_l can be calculated using the following fomula.

$$t_l = \frac{size\ of\ snapshot\ image}{disk\ read\ performance} + t_c$$

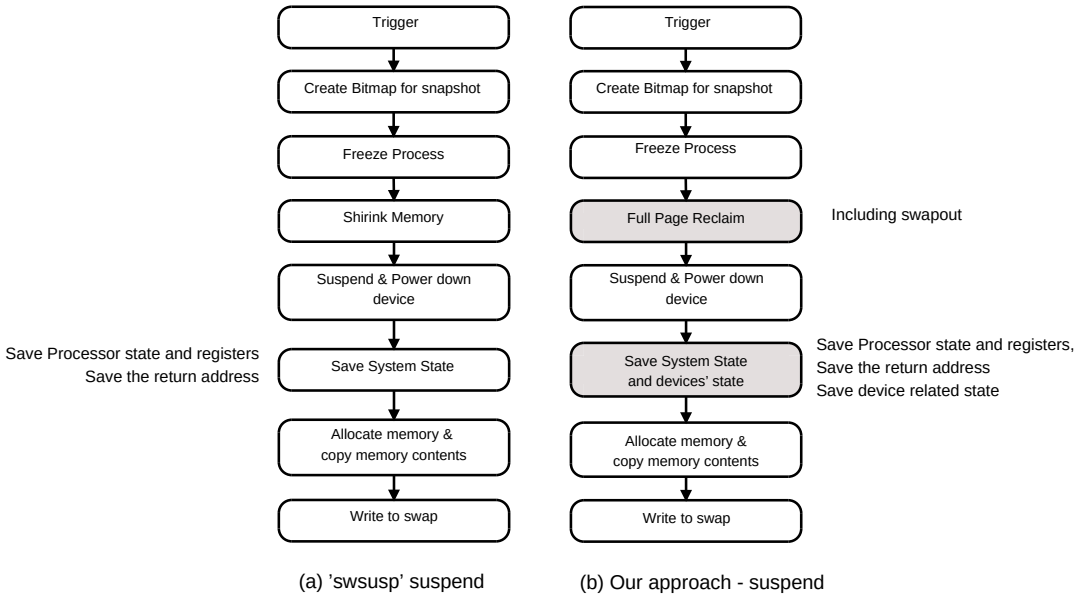

Figure 2: Comparision between the *swsusp* suspend method and our suspend approach

t_c is the time required to copy the loaded snapshot image, stored in in temporal memory, to the assigned memory location. As mentioned above, as the size of snapshot image get bigger, t_l becomes longer. When this happens, we can simply use a compression method such as TuxOnIce to reduce the size of the snapshot image. However, this method requires additional decompression time which increases t_c.

Another factor that influences hibernation-based boot time is t_r which is heavily dependent on the method used to restore the device. In the case of swsusp, all peripherial devices are initialized and then suspended to place them in a resumable state, meaning the same state as when the snapshot image was made. Therefore, if the number of peripherial devices is increased, t_r and t_c are also increased because the memory required for those device drivers is occupied. The snapshot boot technique places peripherial devices into the resumable state and loads snapshot image at the bootloader level. In this way, t_r is much reduced and t_c is eliminated. However, the snapshot boot technique requires additional maintenance costs associated with necessary changes of hardware.

In this paper, we suggest the following two mechanisms to speed up boot times. The first is to minimize the size of the snapshot image in order to reduce snapshot image loading time (t_l) which is the most influential factor hibernation-based boot time. To implement this mechanism, we store snapshot image separately as essential-snapshot-image, which will be loaded at boot time, and supplementary-snapshot-image, which will be restored on demand. The other mechanism is to place the peripherial devices in resumable states using information stored in snapshot image, to reduce $\sum t_r$. The details of these mechanisms are described in the next section.

4 Minimizing Boot Times: Our Approach

4.1 Overall Architecture

In this section, we analyze the suspend/resume flow in swsusp and introduce our improved suspend/resume flow.

As shown in Figure 2, we modify the "shrink memory"[2] stage to "full page frame reclamation," which involves minimizing the size of snapshot image by reclaiming almost of all of the memory required except for essential code and data required for hibernation. At the "save system state" stage, we save information about device related states as well as processor related states to resume the device stage after power up.

As shown in Figure 3-(a), the swsusp resume is started after all devices are initialized. And at the "Suspend device" stage, all of them are suspended – in other words,

[2]A stage in the swsusp suspend flow that ensures enough memory space is allocated to create the snapshot image in memory space.

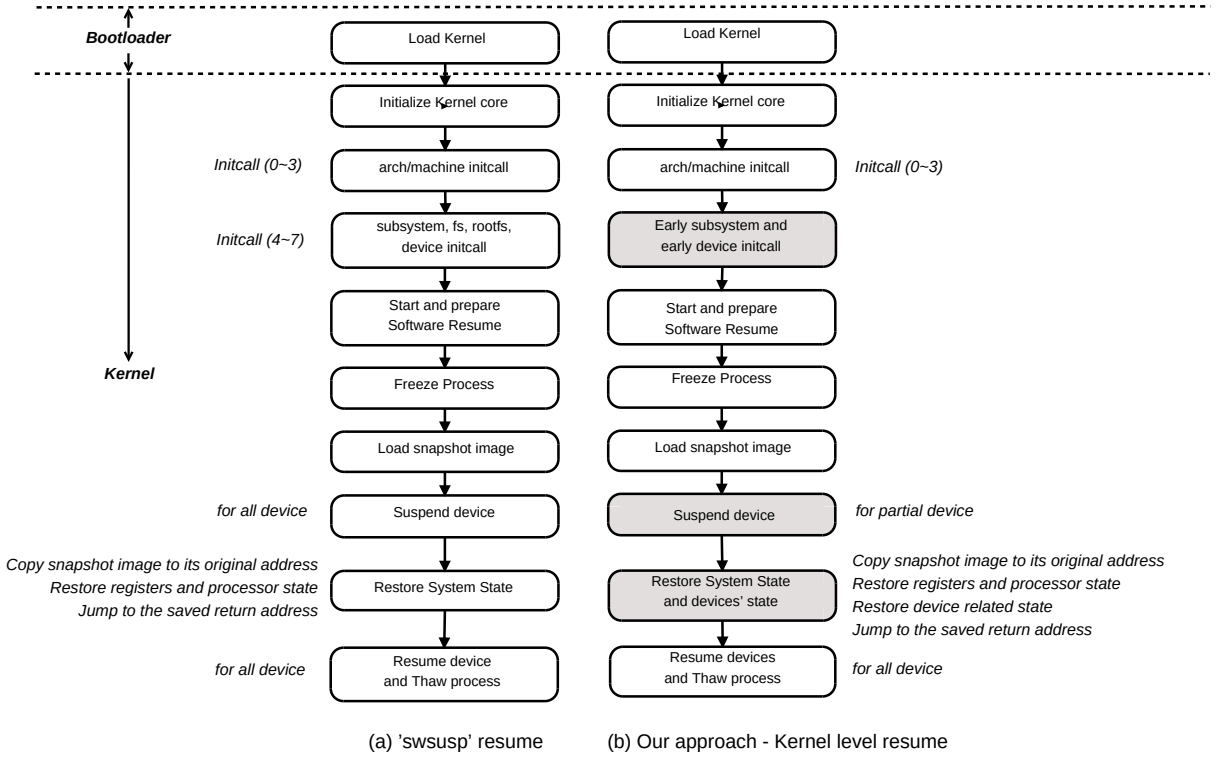

Figure 3: Comparision between the *swsusp* resume approach and our resume approach(kernel level)

this stage place them into the resumable state. Therefore, if there are more devices, or device complexity increases, the time required to initialize and suspend devices will be increased. On the other hand, in our approach as shown in Figure 3-(b), the device initialization and suspend stages are removed and the "restore devices state" is added to the "restore system state" stage. At the "restore devices state" stage, we can place the devices into resumable states based on information that is saved in the snapshot image.

4.2 Full Page Reclamation

Figure 4 outlines a logical view for "full page frame reclamation." Using the "swap out" mechanism in Linux, all application code and data can be reclaimed except locked memory and the caches can be dropped. The reclaimed memory can be restored on demand using the "on demand paging" mechanism in Linux. By taking advantage of these features, the snapshot image can be seperated into two parts, the essential-snapshot-image, which will be restored at boot time, and the supplementary-snapshot-image, which will be restored on demand while the system is running.

To implement the mechanism described above, we create a new swap device for the supplementary-snapshot-image and reclaim pages in the "shrink memory" stage until the number of reclaimable pages reaches zero. We define this mechanism as "full page frame reclamation." As shown in Figure 4, supplementary-snapshot-image, backed up to the file, or dropped.[3] Among the remaining parts, we can exclude the unnecessary parts such as the kernel code[4] using the `register_nosave_region()`. As a result, the essential-snapshot-image includes a minimal number of pages, and we can enter the running state simply by restoring the essential-snapshot-image. Other pages requested by users will be restored on demand by the Linux memory management mechanism.

Smartphones require many processes that must be restored right after boot up, such as idle screens and other service daemons, causing natural delays after boot up. At the moment of boot up, many pages are swapped in for initial running. To improve performance, the supplementary-snapshot-image can be split up and

[3] Before entering hibernation, all pages in the other swap partition must be swapped in

[4] Kernel code is already included in the original kernel image that is loaded by the bootloader.

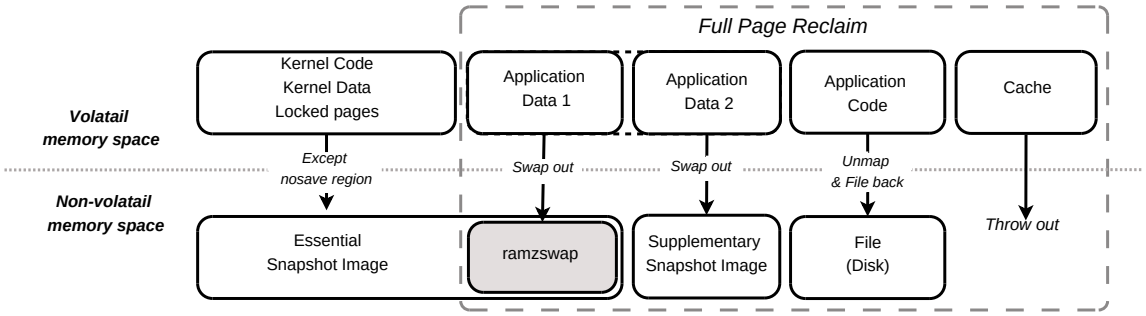

Figure 4: Making a snapshot image

stored in separate swap memory areas: ramzswap[13] and flash/disk swap. Because ramzswap is a RAM based block device, the read performance of ramzswap is better than other swap memory areas. If pages that must be restored right after boot up are stored in the ramzswap area, users only rarely perceive the latency. However, when making the snapshot image, the ramzswap partition is included in the essential-snapshot-image because it is a part of the kernel area. The method chosen to divide the supplementary-snapshot-image is important to improve performance after boot up. An easy method to achieve this is to make the flash/disk swap partition first and the ramzswap partition later. At the "full page reclaim" stage, inactive pages are reclaimed first and active pages are reclaimed later. Therefore, most, or all, inactive pages are stored in the flash/disk swap partition first, and the rest of the pages, including active pages, are stored in the ramzswap partition.

4.3 Fast Device Reactivation

In smartphones, sleep mode, or suspension to RAM (STR), is a necessary implementation because standby time is much longer than actual used time. When a smartphone goes into sleep mode, processes are frozen and devices are suspended to save power. The waking up process reverses the sleep process. A notable characteristic of this mechanism is that the suspended device information is backed up to memory before entering sleep mode, and then restored from memory when woken up by external stimuli.

In our approach, we store suspended device information in the essential-snapshot-image when entering hibernation. This is different from STR because the alive and non-alive block information for the processor are both stored in our approach, while STR stores only non-alive block information. When restoring from hibernation, we place the peripherial devices into resumable state based on the stored information instead of the initializing and suspending stages. However, some devices, including some block subsystems and some block devices that are used for loading snapshot image or devices that require special initializations, should be initialized.

To implement this mechanism, three new initcall sections are added: "early subsystem initcall," "early device initcall," and "resume initcall," as shown in Figure 3. "Early subsystem initcall" and "early device initcall" are required to initialize necessary devices that are used to restore devices from hibernation or to initialize devices that require special initialization. At the "resume initcall" section, the kernel performs the rest of the resume sequence in `software_resume()`.

In fact, the "initall 4∼7" sections[5] are the most time consuming parts of the normal booting sequence because it includes lots of delay routines. If there are many devices, or many kinds of devices, involved, then boot time will be increased. As a result, the time required for those sections is decreased with our mechanism.

Our mechanism operates via simple re-ordering of part of the subsystem/device initializing sequence, so it can continue to work in case of normal boot up, as shown in Figure 5. This mechanism can be used when restoring at the bootloader level, like the snapshot boot technique with a simple modification. When restoring at the kernel level, the technique is more generic and does not require additional management costs. The trade-offs between boot time and management cost can be minimized by manipulating the features of the system. More details about these trade-offs are discussed in section 6.

[5] initcall 4∼7 has "subsystem initcall," "device initcall," and "rootfs/fs initcall."

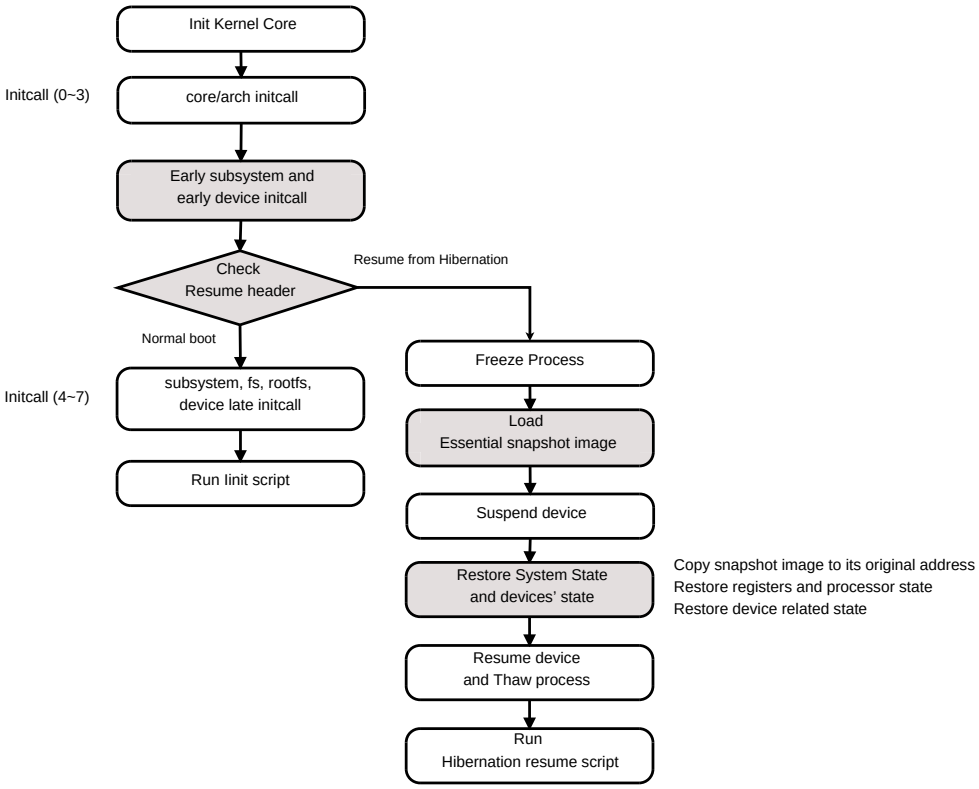

Figure 5: Our mechanism - booting sequence flow

4.4 Resolving Inconsistency Problems

The original purpose of hibernation is not for booting but for restoring, and in such cases the snapshot image is used only once. However, for hibernation-based booting, if the snapshot image is created newly at every boot-up the life of flash memory may decrease due to frequent I/O operation. In addition, it is difficult to produce a snapshot image representing the state of a system right after boot-up, and it takes a long time to do so. Situations of power failure situation must also be considered when entering hibernation.

However, the keep-image mode[6] results in inconsistency problems, because the information in storage can be changed anytime. TuxOnIce recommends following two methods to resolve inconsistency problems. The first is using a read-only file system. The second is to unmount the file system before entering hibernation and re-mount the file system after restoration from hibernation. However, in the real world, such constraints may be unacceptable. So, we tried the other way to resolve this problem. The way is updating superblock[7] and inodes[8] in memory when restoring from hibernation, forbidding to modify inodes included in a snapshot image after restoration from hibernation.

In keep-image mode, SIM[9] or database information inconsistency problems and changes of user configurations must also be considered for implementations. Modem devices use external storage such as SIM card, and information saved in a SIM card, or the SIM card itself, can be changed anytime. Some service daemons like alarm must be reinitialized according to configuration changed by user. Therefore, proper synchronization is required after boot up from hibernation.

5 Experiments

A smartphone based on Linux kernel 2.6.29 is used for this experiment. This smartphone has a Samsung S5PC110 CPU and a Cortex A8 processor, 512MB of Flash memory and 384MB of DRAM. A UBI[15] and

[6] Using the same snapshot image for every boot-up is referred to as keep-image mode in TuxOnIce.

[7] A structure representing the underlying filesystem

[8] The objects that represent the underlying files

[9] Subscriber Identity Module

three UBIFS[16] are used as the file system, and the LCD resolution is 400x800. The I/O performances of the Flash memory are 20MB/s for read and 3MB/s for write. Before the experiment, we implemented swsusp for ARM Cortex A8 because the kernel does not support the software suspend mechanism for ARM. The ACPI code lines were disabled because ARM does not support ACPI.

We make the snapshot image in IDLE screen view right after boot up, and keep it in the swap partition. Before making the snapshot image, we mark some regions as nosave_region to exclude them from the snapshot image, such as the kernel code, a portion of the frame buffer, the sound buffer, and the reserved region for the camera and 3D using `register_nosave_region()`. Every boot up, the same snapshot image is loaded to measure the performance.

5.1 Full Page Reclamation

Table 1: Boot time - Full Page Reclamation

Category		Time(ms)
Bootloader	initialization	597
	kernel image loading[a]	270
	go kernel	27
kernel	Kernel core init[b]	214
	initcall 0 ∼ 3	37
	initcall 4 ∼ 7	3,749
	prepare resume	12
	snapshot image loading[c]	741
	device suspend (all)	236
	copy memory to original	77
	resume device and thaw process	453
Total		**6,413**

[a] Size of kernel image = about 5.5MB
[b] Include 100ms of calibrating delay
[c] Size of snapshot image = 15MB

Before applying "full page frame reclamation", the size of snapshot image is 120MB, and loading time alone takes about 6 seconds. After applying "full page frame reclamation", we obtain a 15MB essential-snapshot-image and a 50MB supplementary-snapshot-image. As a result, we can reduce the size of the snapshot image about 87.4%, and the loading time is dramatically reduced to 0.75 second. We measure the time for each stage using a hardware (H/W) timer, and Table 1 shows the boot time when only "full page frame reclamation" is applied. Total boot time is 6.4 seconds, but there is some delay required in initial operation to load the supplementary-snapshot-image on demand. If 10MB of ramzswap partition is applied, it will require an additional 494ms of boot time to load 10MB of ramzswap partition, but the user will only rarely perceive the latency.

5.2 Fast Device Reactivation

Table 2: Boot time - Full Page Reclamation and Fast Device Reactivation

Category		Time(ms)
Bootloader	initialization	597
	kernel image loading[a]	270
	go kernel	27
Kernel	kernel core init[b]	214
	initcall 0 ∼ 3	37
	early subsystem initcall early module initcall	59
	prepare resume	7
	snapshot image loading[c]	741
	device suspend (partial)	35
	copy memory to original	61
	resume device and thaw process	492
Total		**2,540**

[a] Size of kernel image = about 5.5MB
[b] Include 100ms of calibrating delay
[c] Size of snapshot image = 15MB

According to Table 1, restoring from hibernation is started after 4.894 seconds. The most time-consuming task is initcall 4∼7, because the smartphone used in this experiment includes many peripheral devices. In the "fast device reactivation" technique, we add the "early system init" and "early device init" sections before resume. The "early system init" includes a memory technology device (MTD) and block I/O subsystem initialization, and the "early device init" includes flash device initialization. Initialization of the power management chip is added to the "early device init." As shown in Table 1, restoring from hibernation is started after 1.204 seconds with the "fast device reactivation" technique. As a result, we achieve boot up within 3 seconds when applying both "full page frame reclamation" and "fast device reactivation". The time required for the "device suspend" stage is reduced by about 85%. By extension, we can compare these results with results for restoring the bootloader level. If the snapshot image is loaded at the bootloader level, we can skip some tasks – kernel image loading (270ms), go kernel (27ms), kernel core init

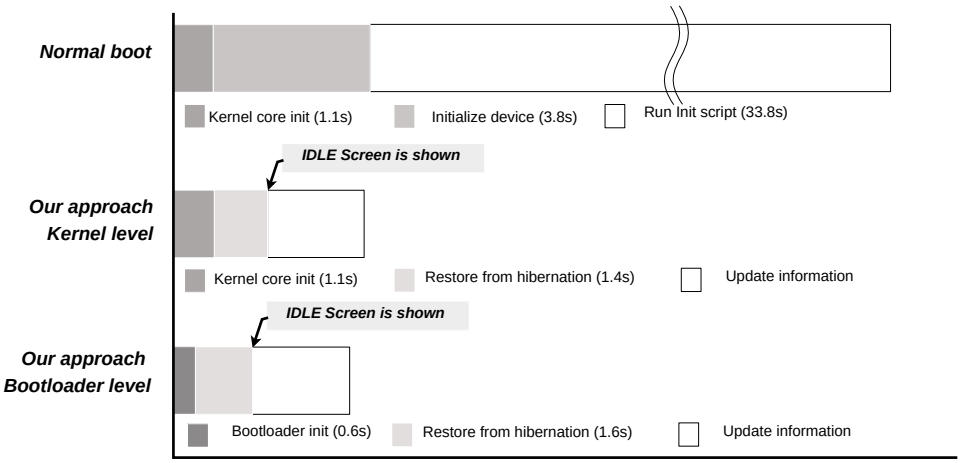

Figure 6: Estimated boot time for each technique

(214ms), initcall 0∼3(37ms), early subsystem/module initcall (59ms), prepare resume (7ms), device suspend (35ms), and copy memory to original (61ms). The total time requied for these tasks is 610ms, not including the time for calibrating the delay (100ms), but we must add 206ms because the loading kernel image includes kernel data as well as kernel code. In other words, the total reduction from loading the snapshot image in the bootloader is under 0.5 second. Further details about applying our approach at the bootloader level are discussed in the next section.

5.3 Resolving Inconsistency Problems

We add a file system recovery stage after boot from hibernation to solve the file system inconsistency problem. The file system recovery stage includes following operations: UBI re-scanning, updating UBIFS superblock in memory, updating inodes in memory. Our current implementation does not include forbidding modifications for inodes which are included in essential-snapshot-image yet, and it is left as our future work. As a result, 2.4 seconds[10] are added after boot from hibernation to recover the file system, but it can be improved by improving the UBI re-scanning method.

To resolve the modem service inconsistency problem, we simply stop the modem service daemon before making the snapshot image. After boot from hibernation, we execute the modem service daemon to synchronize with the modem device. For other service daemons like alarm daemon, we publish an update notification message using inotify to force them to update their information.

[10]The UBI re-scanning operation requires 1.8 seconds for 512MB memory and updating UBIFS superblock requires 0.6 seconds.

5.4 Estimation

Figure 6 shows a comparision of boot times between normal boot mechanisms and our improved mechanisms. While a normal boot takes about 40 seconds, we visualize the idle screen within 3 seconds and total boot time does not exceed 6 seconds with our mechanism. If the approach is applied at the bootloader level, we realize an additional reduction of 0.5 seconds.

6 Discussion

To apply our mechanism at the bootloader level, some functions must be implemented in the bootloader which are already implemented in the kernel: snapshot image loading, initializing some devices, and some other functions. As a result, applying these mechanisms at the bootloader level can eliminate another 0.5 seconds of boot time. Although the bootloader level approach require additional implementation and management, the required works are much less than the snapshot boot. This result suggests that there is a trade-off between boot time and management cost.

7 Conclusions and Future Work

This paper introduce two mechanisms: "full page frame reclamation," which minimizes the sizes of snapshot images, and "fast device reactivation," which improves device reactivation flow. As a result, we designed a platform independent mechanism that can be easily applied

to Linux-based software platforms and eventually obtained instant boot time in a Linux based smartphone. We also considered some issues stemming from keep-image-modes. Obviously, some obstacles still remain to applying these mechanisms to commercial products, such as showing splash, and some other inconsistency problems. However, we believe that these issues may be overcome with proper user workflow and careful verification.

References

[1] Tim R. Bird, "Methods to Improve Bootup Time in Linux," In Proc. of the Linux Symposium, 2004.

[2] A. Leonard Brown, Rafael J. Wysocki, "Suspend-to-RAM in Linux," In Proc. of the Linux Symposium, 2008

[3] Hiroki Kaminaga, "Improving Linux Startup Time Using Software Resume," In Proc. of the Linux Symposium, 2006

[4] Heeseung Jo, Hwanju Kim, Hyun-Gul Roh, and Joonwon Lee, "Improving the Startup Time of Digital TV," IEEE Transactions on Consumer Electronics, Volume 52, Issue 2, May 2009.

[5] CELF - Boot Time, `http://eLinux.org/Boot\_Time`

[6] Uncompress Kernel, `http://elinux.org/Uncompressed\_kernel`

[7] Fast Kernel Decompression, `http://elinux.org/Fast\_Kernel\_Decompression`

[8] Disable console, `http://elinux.org/Disable\_Console`

[9] Preset LPJ, `http://elinux.org/Preset\_LPJ`

[10] Deferred Initcalls, `http://elinux.org/Deferred\_Initcalls`

[11] TuxOnIce (suspend2), `http://www.tuxonice.net/`

[12] Bootchart, `http://www.bootchart.org/`

[13] Ramzswap, `http://code.google.com/p/compcache/`

[14] Android, `http://www.android.com/`

[15] UBI, `http://www.linux-mtd.infradead.org/doc/ubi.html`

[16] UBIFS, `http://www.linux-mtd.infradead.org/doc/ubifs.html`

[17] Samsung, `http://www.samsung.com/global/business/semiconductor/`

[18] UCL, `http://www.oberhumer.com/opensource/ucl/`

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Conference Organizers

Programme Committee

Proceedings Committee