Selective Segment Initialization: Exploiting NVRAM to Reduce Device Startup Latency

Myungsik Kim, Jinchul Shin, and Youjip Won

***Abstract*—We propose selective segment initialization (SSI) to exploit NVRAM to reduce the device startup latency. SSI locates a kernel binary image in byte-addressable NVRAM and boots the system using this image, eliminating the need to load it from storage. SSI also eliminates the process of decompressing and relocating the OS kernel image in embedded Linux system. The key technical ingredients of SSI are precisely identifying the kernel segments where contents are updated in the course of booting and selectively reloading only these sections each time the system reboots. The fresh copy of the sections can be maintained in NVRAM, NAND flash, NOR flash, etc. In our experiment, SSI reduced the size of the kernel binary image loaded from storage into memory by 90% and reduced the overall device startup time by 54%. This approach can be used not only for cold boot (with NVRAM) but also for warm boot, in which the contents of DRAM persist across the system restart.**

***Index Terms*—Embedded linux, fast boot, nonvolatile random access memory (NVRAM), selective segment initialization.**

I. Introduction

REDUCING boot-up time is of critical concern for modern embedded devices. This work exploits byte-addressable NVRAM to reduce the startup latency of modern embedded devices, e.g., smartphones and smart TVs. The recent advancement of NVRAM opens up new avenues for improving the performance [1] and reliability [2] of computing devices. Adopting byte-addressable NVRAM, e.g., FRAM, STT-MRAM, and Phase-Change RAM, makes it possible to provide finer access granularity in existing block devices [3] and to provide persistence in existing memory subsystems [4]. This study focuses on reducing the startup latency using NVRAM, a newly emerging memory device. The key technical ingredient in this approach is the handling of the kernel's writable sections. In legacy computing paradigm, some sections are implicitly initialized by the loader, e.g., the data section (`.data`), and some sections are explicitly initialized by the program itself, e.g., global variables with initial values (`.bss`). Since most sections of a kernel image are read-only, locating the kernel image in the byte-addressable NVRAM would eliminate the process of loading and relocating the kernel binary image.

When the kernel image resides in NVRAM, the state of each segment will persist and the segments maintain their data in a persistent manner. When the kernel reboots with its variables retaining their previous values, the system can enter an undefined state, eventually causing deadlock or some other malfunction. This study identifies the segments that have been "implicitly" initialized in the legacy startup phase and develops a technique called selective segment initialization (SSI) to explicitly initialize only those segments. In SSI method, a clean copy of the morphable sections is maintained separately. When booting, the SSI overwrites the morphable segments with their clean copies, bypassing the OS kernel loading process. This approach eliminates the process of loading and decompressing the kernel image while maintaining the contents of the NVRAM during startup.

Manuscript received January 08, 2014; accepted February 22, 2014. Date of publication March 14, 2014; date of current version May 23, 2014. This work was supported by the IT R&D Program MKE/KEIT (10041608 , Embedded system Software for New-memory based Smart Device). This manuscript was recommended for publication by T.-W. Kuo.

M. Kim is with the Division of Computer Sciences and Engineering, Hanyang University, Seoul 133-791, Korea, on leave from Digital Appliances Division, Samsung Electronics, Suwon, Korea (e-mail: mskim77@hanyang.ac.kr).

J. Shin was with Hanyang University, Seoul 133-791, Korea, he is now with SK Planet, Seoul 133-791, Korea (e-mail: dawnsea@sk.com).

Y. Won is with the Divison of Computer Sciences and Engineering, Hanyang University, Seoul 133-791, Korea (e-mail: yjwon@hanyang.ac.kr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2014.2312017

II. Related Work

There are a number of approaches for reducing the startup latency of computing devices [5]. Snapshot retains a dump image of memory in the device's storage and loads this image during startup to initialize the device [6]. The problem comes from snapshot size: the larger the dump image, the longer it takes to execute snapshot. Execute-in-place (XIP) is a technique that eliminates the time required to load a binary image from storage by locating it in byte-readable nonvolatile memory (e.g., NOR flash) and executing it directly from that location. XIP boot executes the program directly from where it is stored, reducing boot-up latency. However, while this approach reduces startup time, the overall performance of the OS decreases because NOR is slower than DRAM. NOR flash also has higher cost per bit than NAND flash. Kexec is a rebooting technique whereby the operating system directly reloads the kernel image, bypassing the hardware initialization process [7]. Recently, a number of techniques have been proposed for reducing the startup time of embedded devices. Jo *et al.* applied snapshot booting to Smart TVs [8]. They proposed executing two sets of booting tasks concurrently so that tasks with different resource consumption characteristics are executed in parallel [9]. Baik *et al.* categorized snapshot images for Smart TVs into two groups: the essential-snapshot-image, which is required to initialize the device, and the add-on-image, which can be loaded on demand [10]. SSI complements the existing efforts. None of the existing works distinguishes the read/write characteristics of individual kernel segments nor treats them differently for NVRAM.

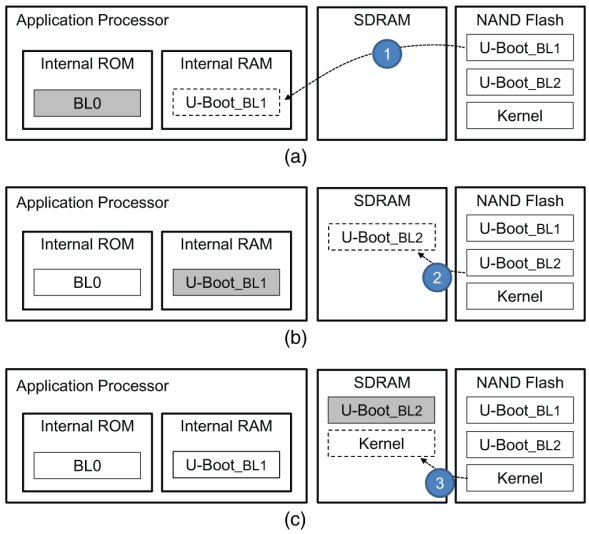

Fig. 1. Image copying phase of boot-up process in S5PC100 processor. (a) Execute BL0 in Iinternal ROM, copying U-Boot_BL1 from NAND to internal RAM. (b) Execute U-Boot_BL1 in internal RAM, copying U-Boot_BL2 from NAND to SDRAM. (c) Execute U-Boot_BL2 in SDRAM, copying kernel from NAND to SDRAM.

III. Background

A. Starting Up a Device: Hardware Initialization

The system boot-up process in embedded systems consists of switching the power on, loading the boot loader, loading the OS kernel from storage, and transferring control to the kernel. Fig. 1 illustrates booting process of mobile application processor [11], where boot loader and kernel are stored in NAND flash. First, the micro boot loader (U-Boot_BL1) at NAND flash is loaded into internal RAM [see Fig. 1(a)]. The built-in 32 KB internal ROM code (BL0) downloads the micro boot loader code from the initial 16 KB of NAND into the internal RAM when NAND boot mode is selected. The micro boot loader is required because the internal RAM is too small (96 KB) to accommodate the entire boot loader. The micro boot loader then fetches the main body of the U-Boot code into DRAM [see Fig. 1(b)]. Afterwards, the main boot loader initializes the hardware to load the kernel into DRAM [see Fig. 1(c)].

B. Loading the Kernel Image

When the boot loader finishes initializing the hardware, it loads the OS kernel image into memory. For an embedded system, a binary image of the Linux kernel is normally stored in compressed form, named `zImage`. Fig. 2 shows the structure of a compressed kernel which consists of three components: `piggy.gz`, `misc.o` and `head.o`. `piggy.gz` is the compressed binary image of the OS kernel; `misc.o` is the decompression module that is used to decompress `piggy.gz`; and `head.o` contains initialization instructions at boot-up time. To boot the kernel, the compressed image is first loaded into main memory by the boot loader. Control is then passed to head.o, which initializes the CPU. `misc.o` decompresses the kernel binary image, `piggy.gz`. The decompressed kernel image is then relocated to the appropriate memory address. At this point, control passes to the relocated kernel, which then initializes kernel objects, loads device drivers for various devices, initializes peripherals, and starts the user service.

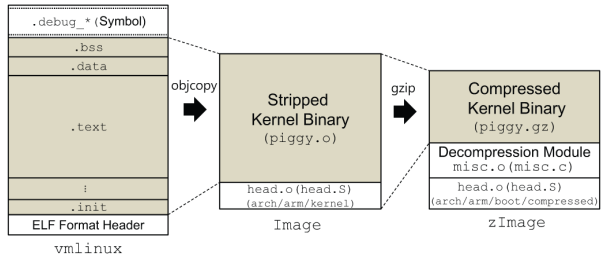

Fig. 2. Making compressed kernel binary from executable kernel image.

IV. Selective Segment Initialization

A. Motivation

It is generally accepted that NOR flash is used for code fetching memory in lightweight embedded systems. NOR-XIP is a suitable solution for microcontroller because it can be a simple memory model that requires small amount of RAM [12]. However, NOR flash has a performance drawback, which is slow access time compared to DRAM. Therefore, a modern embedded Linux system exploits store and download (SnD) method with NAND (or eMMC) and DRAM instead of NOR-XIP. SnD model uses a compressed image to minimize memory footprint in a storage memory, which requires image loading and decompression when booting. SnD can provide faster instruction fetching speeds because the program code resides in DRAM called “shadowing”. Byte-addressable NVRAM has benefits such as harboring data in a persistent manner with access latency comparable to that of DRAM, but NVRAM is still an emerging technology and it may be difficult to replace DRAM with NVRAM in the near future. Therefore, this study explores a compromise on booting scheme even change DRAM to NVRAM still having the benefit of SnD method. A new scheme follows suit with conventional SnD method without significantly changing the memory organization in order to maximize usability. This way, we can maintain the benefits of SnD methods such as memory space saving by using compressed image and data integrity checking by decompression also. The objective of this work is to relocate kernel image to byte-addressable NVRAM and to reuse the relocated kernel image. Thus, we eliminate the overhead of loading, decompressing, and relocating the OS kernel.

B. Problem Assessment

While maintaining the kernel binary image in NVRAM eliminates a significant fraction of the entire booting procedure, such as loading, it raises new issues that require elaborate treatment. In legacy booting procedure, some kernel sections are implicitly initialized when the OS is loaded from storage. If the kernel is maintained persistently in NVRAM removing loading steps at warm-boot procedure, these sections are initialized only when the OS is loaded from the storage to NVRAM. In subsequent warm-booting, the OS kernel starts to execute from the previously stored state of these sections. This may lead the system to enter an undefined state.

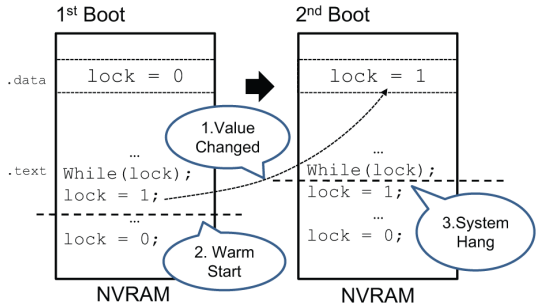

Fig. 3. Example of boot failure in NVRAM.

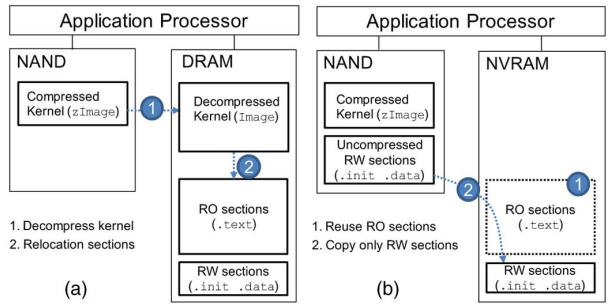

Fig. 4. Startup procedures in legacy and SSI boot. (a) Legacy boot. (b) SSI boot.

TABLE I
SECTION ANALYSIS OF THE KERNEL IMAGE

Section	Attributes	Address (Hex)	Size (KB)
.text.head	CONTENTS, ALLOC, LOAD, **READONLY**, CODE	c0008000	1.0
.init	CONTENTS, ALLOC, LOAD, CODE	c00083e0	167.0
.text	CONTENTS, ALLOC, LOAD, **READONLY**, CODE	c0032000	5008.2
.text.init	CONTENTS, ALLOC, LOAD, **READONLY**, CODE	c05160e4	0.2
__ksymtab	CONTENTS, ALLOC, LOAD, **READONLY**, DATA	c0517000	19.8
__ksymtab_gpl	CONTENTS, ALLOC, LOAD, **READONLY**, DATA	c051bf40	7.9
_ksymtab_strings	CONTENTS, ALLOC, LOAD, **READONLY**, DATA	c051dea8	60.0
__param	CONTENTS, ALLOC, LOAD, **READONLY**, DATA	c052ce80	4.4
.data	CONTENTS, ALLOC, LOAD, DATA	c052e000	164.1
.init.rodata	CONTENTS, ALLOC, LOAD, **READONLY**, CODE	c0557040	0.3
.bss	ALLOC	c0557140	248.2

Fig. 3 illustrates this situation, `lock` is a global variable used to protect the critical section. It resides in `.data` and is initialized to 0 at compile time. When the kernel is loaded into memory for the first time and `lock` equals 0, the program exits a while loop controlled by this logical value. As the kernel executes, it eventually sets the value of `lock` to 1. If the kernel is restarted when the `lock` variable in NVRAM retains the value of 1, the boot process will not be able to exit the while loop.

C. Indentifying Reusable Kernel Segments

One of the key ingredients of SSI is to determine the reusability of individual sections. We analyzed the ELF image of a Linux kernel and determined which portion of memory address in NVRAM needs to be reset and which portion can be reused. `objdump` was used to analyze the section organization of the kernel image. Table I shows the result: all sections except `.init`, `.data`, and `.bss` are read-only attribute. `.init.rodata` contains the constant variables used for the OS kernel initialization. A separate section (`.bss`) contains global and static variables and a code segment that explicitly initializes these variables. SSI uses the section map to partition the kernel image and to determine the location (DRAM versus NVRAM) of the individual sections.

D. Selective Segment Initialization

The legacy computing paradigm separates the notion of "memory" and "storage" and explicitly defines the tasks to be done in the "loading" phase, e.g., initialization of a certain section. Compiler generates the binary so that a certain section of the image is initialized each time it is loaded from storage to memory. However, when "load" phase does not exist, modifications are needed in all or some of compiler, linker, loader, and operating system for ELF based binary file to be still legitimate. An important constraint is to minimize the changes in the existing software development tools, e.g., compiler, loader, linker, and OS to use NVRAM to harbor OS kernel. In this work, we identify the sections in ELF that cannot be reused and explicitly initialize these sections each time the OS kernel starts. We call this technique SSI. A clean copy of each morphable section can be stored in byte addressable NVRAM, or in a block device, e.g., NAND based storage. In booting with SSI, we identify the sections with read/write (RW) permission, maintain the clean copy of RW sections in a separate location, and load (or initialize) only the RW sections. When the clean copy of read-only (RO) sections remain in NVRAM, boot loader loads only RW sections and therefore significantly reduces the time to load, decompress and relocate the whole kernel segments. This work can be used not only in NVRAM enabled systems but also in legacy DRAM systems where DRAM contents remain at warm boot session.

Fig. 4 shows differences between legacy and SSI startup procedures. In SSI restart, a fresh version of `.data` and `.init` sections are explicitly initialized each time the OS kernel restarts. According to our experiment, the compressed image of the OS kernel was 3164 KB, whereas the combined size of the uncompressed `.init` and `.data` sections corresponded to only 331 KB. Thus, SSI can reduce the amount of data to be loaded from NAND storage to 1/10.

V. IMPLEMENTATION & EXPERIMENT

A. Implementation of Dual Mode Booting

At startup, the boot loader decides whether to "Cold Boot" or "SSI Boot." Cold boot occurs if it is explicitly chosen or if a kernel image does not exist in NVRAM. The start address and the size information are verified by examining the location of the symbols in each section. SSI boot uses the uncompressed `.data` and `.init` sections and links them with the existing kernel sections in memory. SSI can also be used in soft-reset of commodity smartphones. For this approach, a hot-key for soft-reset and a dedicated device driver for SSI Boot were developed in an existing smartphone platform. Pseudocode 1 shows the implementation of SSI boot function. For cold boot, the entire kernel image is loaded from storage into memory and is initialized. With SSI boot, copying and decompressing of kernel image phases can be eliminated by maintaining a kernel image in NVRAM. When a reboot is triggered during the booting process or when the kernel reboots due to a system crash, the boot loader is set to perform a cold boot.

Pseudocode 1 Dual Mode Booting for SSI Boot

```
1: Start U-Boot
2: if Reset Status == Power Reset then ▷ Cold Boot mode
3:     Copy compressed kernel image (zImage)
4:     Decompress kernel image
5: else ▷ SSI Boot mode, Reset Status == Soft Reset
6:     Copy uncompressed RW section (.init, .data)
7: end if
8: Start kernel_start()
```

Evaluation was conducted in a smartphone development board. The board has S5PC100 ARM Cortex-A8 based Mobile Application Processor at 667 MHz, 128 MB of NAND flash, and 512 MB of DDR2-SDRAM. We implemented SSI Boot scheme into the U-Boot firmware at early boot stage on Linux kernel 2.6.29. For SSI enabled booting, the modified U-Boot loads the clean copies of .init and .data sections from NAND flash memory and overwrites the respective sections with the clean version to NVRAM. The read-only sections and the. bss section, which have explicit initialization mechanisms, are reused. We used software reset and SDRAM to emulate NVRAM. When the software reset is pressed, the SSI enabled boot loader executes the kernel image in NVRAM (emulated with SDRAM) after it initializes the .data and .init sections. The software reset was implemented by the GPIO device driver. If the reset button is pressed, the button driver will generate the software reset by writing 0xc100 at SWRESET register. When the software reset is triggered, the Program Counter jumps to the system reset vector and U-Boot firmware starts. U-Boot was modified to identify the type of booting. If the reset is a power-on-reset, U-Boot will load the full kernel image. Otherwise, U-Boot will load only .data and .init sections. The time spent at each boot stage was measured by the internal PWM timer no. 3 in S5PC100 processor. The PWM timer was initialized with 1 ms resolution at the beginning of the U-Boot. It measured the time without running U-Boot or kernel debug service.

B. Results

The size of the compressed kernel image (zImage) was 3164 KB. The total size of morphable segments (.data and .init) corresponded to 331 KB. With SSI, the amount of data loaded from storage decreases by 90% from 3164 to 331 KB. Table II and Fig. 5 shows the system boot times for cold boot and SSI boot. The time was measured from power-on to the end of the boot script. With SSI boot, the boot time decreased by 54% compared to cold boot. The time for U-Boot was the same for cold boot and SSI boot. The time for loading the kernel image was 1459 and 175 ms for cold boot and SSI boot, respectively. SSI reduced the kernel copying time by reusing read-only sections that previously resided in NVRAM. In addition, SSI boot does not need the kernel decompression time.

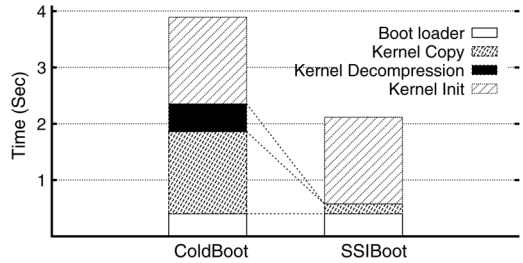

Fig. 5. Comparison of startup time in cold boot and SSI boot.

TABLE II
TIME MEASUREMENT OF KERNEL BOOT

Step	Cold Boot	SSI Boot
Boot loader (U-boot)	402 ms	402 ms
Kernel Copy	1459 ms	175 ms
Kernel Decompression	491 ms	0 ms
Kernel Init	1540 ms	1540 ms
Total Time	3892 ms	2117 ms
Time Reduction in %	N/A	54%

VI. CONCLUSION

A new startup mechanism, called SSI, was proposed to exploit byte-addressable NVRAM to improve the device start-up latency. In SSI, reusable kernel sections are maintained in NVRAM and morphable kernel sections are selectively initialized in an explicit manner. This approach enables the legacy system to exploit the nonvolatility of byte-addressable NVRAM without significantly modifying the existing software stack. We implemented SSI in an existing smartphone platform and achieved 54% reduction in overall startup latency of Linux kernel.

REFERENCES

[1] Y. Xie, "Modeling, architecture, and applications for emerging memory technologies," *IEEE Design Test Comput.*, vol. 28, no. 1, pp. 44–51, Jan.-Feb. 2011.
[2] A. Makarov, V. Sverdlov, and S. Selberherr, "Emerging memory technologies: Trends, challenges, and modeling methods," *Microelectron. Rel.*, vol. 52, no. 4, pp. 628–634, 2012.
[3] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, New York, NY, USA, 2009, pp. 133–146, ser. SOSP '09, ACM.
[4] A. Badam, "How persistent memory will change software systems," *Computer*, vol. 46, no. 8, pp. 45–51, 2013.
[5] C. Hallinan, "Reducing boot time in embedded linux systems," *Linux J.*, vol. 2009, no. 188, p. 4, 2009.
[6] H. Kaminaga, "Improving linux startup time using software resume (and other techniques)," in *Proc. Linux Symp.*, 2006, p. 17.
[7] A. Pfiffer, "Whitepaper: Reducing system reboot time with kexec," devresources. linuxfoundation. vol. 4, 2003.
[8] H. Jo, H. Kim, H.-G. Roh, and J. Lee, "Improving the startup time of digital tv," *IEEE Trans. Consumer Electron.*, vol. 55, no. 2, pp. 721–727, May 2009.
[9] H. Jo, H. Kim, J. Jeong, J. Lee, and S. Maeng, "Optimizing the startup time of embedded systems: A case study of digital tv," *IEEE Trans. Consumer Electron.*, vol. 55, no. 4, pp. 2242–2247, Nov. 2009.
[10] K. Baik, S. Kim, S. Woo, and J. Choi, "Boosting up embedded linux device: Experience on linux-based smartphone," in *Proc. Linux Symp.*, 2010, pp. 10–18.
[11] *"S5PC100 USER'S MANUAL REV 1.02,"* Samsung Electronics Co. Ltd., 2009, p. 2.6.
[12] T. Benavides, J. Treon, J. Hulbert, and W. Chang, "The enabling of an execute-in-place architecture to reduce the embedded system memory footprint and boot time," *J. Comput.*, vol. 3, no. 1, pp. 79–89, 2008.