Bootless Boot: Reducing Device Boot Latency with Byte Addressable NVRAM

Dokeun Lee
Hanyang University, Korea

Youjip Won
Hanyang University, Korea

Abstract—**Booting is an essential process that loads an operating system (OS) to a main memory and initializes all the system. In this paper, we propose a new technique to shorten the boot-up time by using a non-volatile random access memory (NVRAM). This technique eliminates the kernel loading, the kernel decompression, and the reallocation of a kernel image by maintaining of the linux binary image at the non-volatile random access memory. The essence of locating OS binary image at NVRAM lies in how to re-initialize the segments that are updated as OS executes and have been initialized implicitly while OS is loaded onto memory from the storage device. We develop a Bootless Boot which completely removes the process of loading any of the kernel images. Bootless Boot consists of (i) object filter which identifies the kernel objects which need to be initialized every time OS boots up, (ii) explicit initialization module which explicitly re-initializes the objects identified by the object filter. Explicit initialization module is added to the linux kernel.**

I. Introduction

The word ”Booting” comes from ”Bootstrapping” which used as a metaphor for self-sustaining process that can proceed without help[25]. These days, not only the legacy computers, e.g. PC and Servers, but also mobile devices, e.g. smartphone, smartpad, and etc. and home appliances, e.g., TV and refrigerator, use general purpose OS, e.g. Linux/Android, to control its hardware and to run various application[3]. Booting the device is a complex and complicated procedure. Numerous efforts have been put to reduce the booting latency[14][5][13][12][1][24][26] . Boot latency is one of the key factors which governs the quality of the device, especially the embedded one, e.g. TV[7][19].

Primary task of booting is to load the binary image of OS from the storage device to a main memory and to execute OS to initialize the state of the system so that it is ready to serve the user request. The *loading* consists of loading the compressed binary image of OS, decompressing the compressed image and relocating and linking the decompressed image. The loading, decompressing, and linking/relocation procedure of starting an OS is due to the separate notion of `memory`, to and from which the CPU stores and loads the data and instructions and *storage*, where the program and data resides.

The rapid advancement of semiconductor technology makes the new type of memory which can hold data without electricity and be accessed similar to legacy DRAM commercially available in the foreseeable future. They include Phase-change RAM (PRAM)[18], Spin Transfer Torque-Magnetic RAM (STT-MRAM)[9], Ferroelectric RAM (FeRAM)[6] and etc. Early versions of these devices have already been deployed in the market, e.g. tag-memory[21], NAND based storage device[2], and small storage for sensor device[27]. A fair amount of efforts has been proposed to devise a new software layer to effectively exploit these memory devices in the legacy computer system which consists of CPU, memory and storage.

In this work, we aim at reducing the device startup latency, *booting* exploiting the byte-addressable NVRAM. We develop a technique, *Bootless Boot*, where the device maintains binary image of OS at byte-addressable NVRAM in a decompressed and pre-linked form. Different from the DRAM, the NVRAM maintains the state of memory across the power-reset cycles. What lies at the core of Bootless Boot is to restore the initial value of various kernel objects when the device needs to startup, i.e. to execute the fresh copy of the OS. In Bootless Boot, we avoid using the ”memory-copy” technique in restoring the initial value of kernel objects since the memory-copy is an expensive operation especially in an embedded device. Bootless Boot consists of *kernel object filtering module* which scans the kernel source code and identifies the kernel objects which are initialized at the object declaration, and *object initialization module* which explicitly initializes the kernel objects obtained from object filter. Kernel object filter scans the OS source code and identifies the initialized global and static variables, i.e. which resides in `.data` section. Bootless Boot has a source code generator which accepts the name, type, and initial value of the kernel variable(object) found by the object filter, and outputs the source code which performs the initialization of the source code.

II. Background

A. Booting of Computing Device

Booting sequence is a process which initializes CPU, memory and other peripheral devices to make a perfect circumstance to execute the operation system[7]. While the system boots up, such works as initializing hardware, and loading up the OS to the main memory are carried out to operate the system.

Basic Input Output System (BIOS)[23] in Read-Only Memory (ROM) operates with the power on in the common PC environment. Encompassing three stages – a start-up step, a system initiative step, and a bootloader loading step, the BIOS is a kind of firmware to initialize and to test the basic hardware. Undergirded by the Power-on Self Test (POST) process in the start-up step, the BIOS tests the basic hardware. It then initializes the hardware to examine the types of

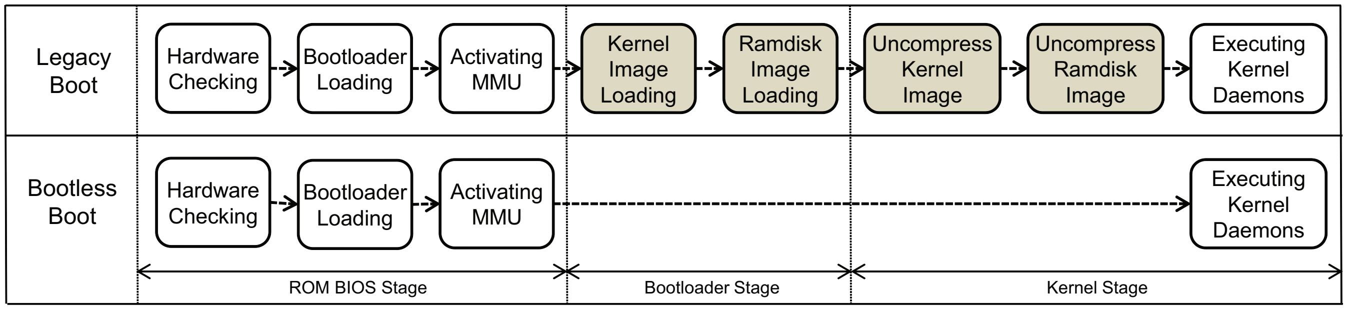


Fig. 1. Bootless Boot Process Compared with Legacy Boot Process

connected devices in the system initiative step, ensued by loading a bootloader code to a main memory in the last stage. As far as the BIOS is concerned, a bootloader takes charge of a kernel binary image loading only. In an embedded system environment which has little to do with the common PC environment, a bootloader such as uboot[4] is used instead of the BIOS as a firmware to check the basic hardware and initialize the CPU clock, memory, and other hardware, except for those indirectly related with the boot-up process such as LCD screen or sound-related devices. In the last phase, it loads the kernel binary image from the secondary storage to a main memory.

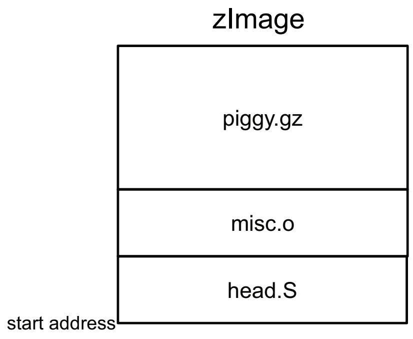
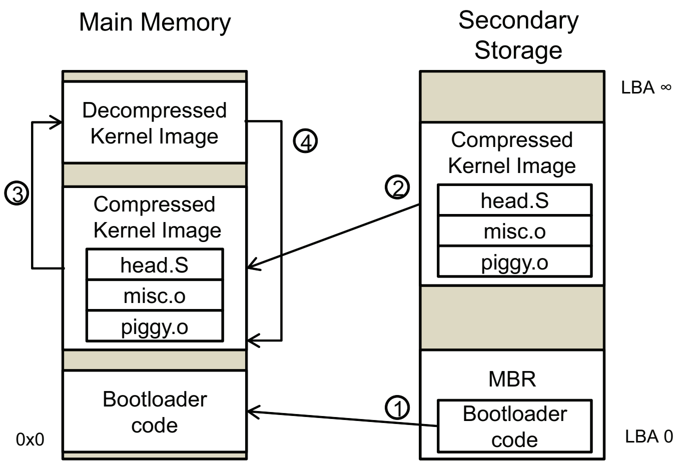


Fig. 2. Kernel zImage Structure

A generally used OS kernel image form is zImage (Fig.2) in an embedded system environment. The zImage consists of three binary files: (i) *head.S*, (ii) *misc.o*, (iii) *piggy.o*. *head.S* is an collection of boot-up codes, rendering Memory Management Unit (MMU) to translate virtual address into physical address, initializing `.bss` section, which has global objects with no initial state, and reallocating the decompressed kernel to kernel start address. *misc.o* is a decompressing code for compressed kernel images, and *piggy.o* is a compressed main body of kernel. A bootloader loads a kernel image to predefined physical address. The target address is specified in the linker script. When the bootloader finishes loading of the kernel, the control is transferred to the beginning of the OS kernel, *head.S* which accordingly calls the decompressing function in the *misc.o* to decompress *piggy.o*. It then decompresses the compressed kernel image to the predefined address. Because the decompressed kernel is a linked program code which must be placed at a fixed location, it must be reallocated to the starting address as described in Fig.3.



1. The BIOS loads the bootloader to main memory.
2. The bootloader loads the compressed kernel image to the main memory.
3. The decompression codes in the misc.o decompress the compressed kernel image.
4. The decompressed kernel is reallocated to compressed kernel's position for space efficiency

Fig. 3. Kernel Image Decompression and Reallocation

This progress also secures the memory space by eliminating the compressed kernel image that is no longer in use. The main purpose of *head.S* consists in making a special register which is related with paging to a specific state for using page table. For example, Intel x86 CPU has CR0 register to activate the MMU. If the most significant bit (MSB) of the CR0 register is set to 1, the MMU will activate, and then start to translate the virtual address into physical address by referring to the page table. The CPU can access a main memory directly through the physical address in advance to this process, but once the process is done, it can only access the main memory through the virtual address. The last step of 'head.S' is to initialize the .bss secion to zero and to jump to the `start_kernel()` function.

At this point in time, the OS initializes date structures that are related to the CPU, the page table, the slab, the scheduler, the timer, the drives, and etc. After the initialization

Item	DRAM	NAND Flash	FRAM	MRAM	STT-MRAM	PRAM	ReRAM
Present Density	8GB/Chip	64GB/Chip	128Mb/Chip	32MB/Chip	2MB/Chip	512MB/Chip	64KB/Chip
Cell Size(SLC)	$6F^2$	$4F^2$	$6F^2$	$20F^2$	$4F^2$	$5F^2$	$6F^2$
MLC Capability	No	4Bits/Cell	No	2Bits/Cell	4Bits/Cell	4Bits/Cell	2Bits/cell
Program Energy/Bit	2pJ	10nJ	2pJ	120pJ	0.02pJ	100pJ	2pJ
Access Time(W/R)	10/10ns	200/25us	50/75ns	12/12ns	10/10ns	100/20ns	10/20ns
Endurance Retantion	10^{16}/64ms	10^5/10yr	10^{15}/10yr	10^{16}/10yr	10^{16}/10yr	10^5/10yr	10^6/10yr

TABLE I
NON-VOLATILE MEMORY CHARACTERISTICS [16]

of these data structures, the `start_kernel()` function calls the `rest_init()` function to make the 'init' process to take charge in the execution of user programs such as kernel daemons, shell, and etc., followed by calling the `cpu_idle()` function to stay on standby at the infinite loop. This becomes an idle process as `start_kernel()` generally stays inactivated yet starts to operate when there is no process in the OS scheduler to execute. The init process calls functions which are registered by `initcall` macro and kernel daemons in the */sbin/init*. During this process, the shell program is loaded to provide the interface between users and the OS.

B. Non-volatile Random Access Memories

NVRAM is a new type of memory which, unlike DRAM, can save its information permanently without the power supply, while entailing a similarity to the formerly mentioned memory that it can operate program codes[15]. Therefore, it is possible to be used as a main memory. Non-volatile memories such as PRAM, FeRAM, STT-MRAM and Resistive RAM(ReRAM)[20] have been recently studied by many researchers. They are at the same level of DRAM in reading/writing speed, but they have not yet outstripped it as they have distinct weaknesses such as present density and endurance. However, NVRAMs have been drawing attention as a next generation's main memory because they can achieve a large scale, low power consumption, in comparison with DRAM which has structural limitation.

C. Embedded Linux

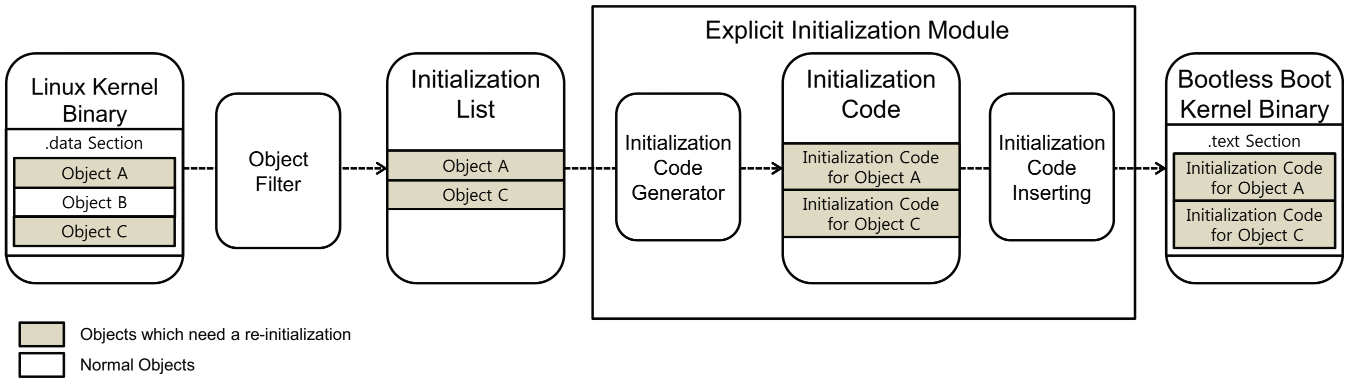
Embedded Linux is used for devices with special purposes such as TV and mobile phones, rather than usual computing system. Firmware used to operate such devices in lieu of heavy operating system[8], but due to the tendency change along with *digital conversions* and *ubiquitous*, operating systems are required because of their functionality in multiple roles. Aforementioned devices have a very limited number of resources compared to usual computing system, so general linux is not appropriate for them. Hence, it is better to use an embedded linux which is optimized for such devices. Generally, an embedded linux is used for machinery with special purposes such as information appliances and industrial machines. Such machines have to pass their control to system users quickly[10][1][11]. If a mobile phone takes a few minutes to use after the booting, it will be sold poorly in the market. Also industrial machines require fast boot up as soon as they are turned on because boot-up time affects the amount of output produced. By the year of 2011, the boot-up time of a smart phone took at least 20 sec[19]. If the boot-up time can be reduced, it will be considered as quality, user friendly product. In an embedded system, DRAM is usually used for a main memory while NAND flash is used for storage[16]. Hard disk is not preferred in mobile or portable devices because it is sensitive to physical impact and not appropriate for device miniaturization. NVRAM can be used as a main memory because it is byte-addressable and also can be used as storage because it contains memory cell information even in the power-off state. They have an advantage in low-power requirement and scalability. Therefore, it is an appropriate memory for an embedded system even though it may not excel in writing speed and durability.

III. RELATED WORKS

There are two kinds of techniques to improve the boot-up speed: the first group is optimization of each step in the boot process, whereas the second group bypasses some steps of the boot process. The first group of these techniques modifies each step in the boot process to execute only essential works or eliminate some modules of kernel[14][5][13]. They particularly optimize the init process intensively because the init process accounts for the most of the boot-up time. The techniques such as *initng* and *upstart* are well-known among groups with similar techniques. The second group of these techniques uses hibernation or shapshot[22]. Fuji developed a fast boot technique for commercial phones by using hibernation, and Joe, Baik, Wang and Yamakita used snapshot for fast booting[12][1][24][26].

IV. DESIGN

In designing Bootless Boot, our goals are to remove the loading process of kernel and ramdisk image as well as the decompressing process of these images. With an aim of fully achieving the stated goals, Bootless Boot provides an object filter to identify the kernel object that is needed to be initialized when the OS boots up, and an explicit initialization module to clearly initialize the objects identified by the object filter. Fig.4



Linux Kernel Binary
.data Section
Object A
Object B
Object C
Object Filter
Initialization List
Object A
Object C
Explicit Initialization Module
Initialization Code Generator
Initialization Code
Initialization Code for Object A
Initialization Code for Object C
Initialization Code Inserting
Bootless Boot Kernel Binary
.text Section
Initialization Code for Object A
Initialization Code for Object C
Objects which need a re-initialization
Normal Objects

Fig. 5. Design Concept of Bootless Boot

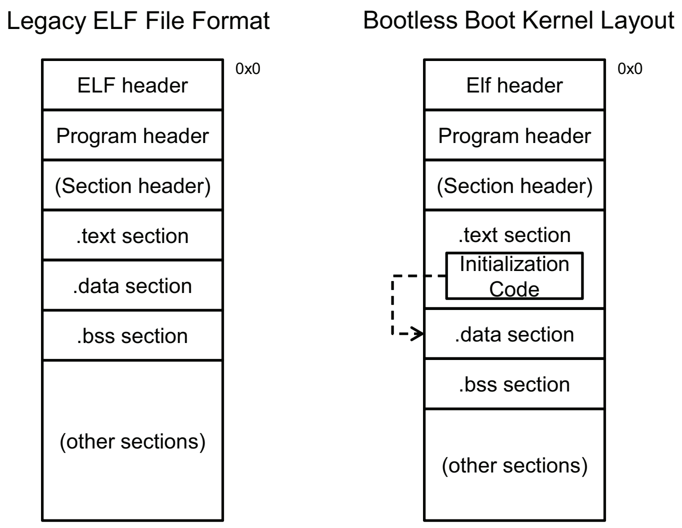


Fig. 4. ELF File Format and Bootless Boot Kernel Layout

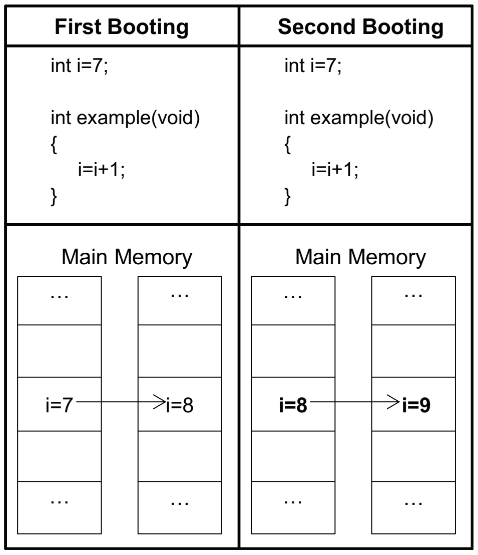
shows the difference in kernel layout between Bootless Boot kernel and legacy kernel. A key feature of Bootless Boot is that it gives the object initialization code to `.text` section instead of kernel loading while eliminating the loading overhead time of the kernel image when the OS boots up and significantly increasing the performance. Fig.5 clearly shows these key features of Bootless boot. We have given a description below on the object filter and the explicit initialization module to elucidate the differences between Bootless Boot and legacy boot-up.

A. Object Filter

Removing kernel loading from legacy boot-up process presents multiple challenges. The kernel should not only protect its binary from memory clear function in the boot-up process, but also re-initialize polluted objects that have previously been used. Potential problems may arise if one object is used at every boot up without appropriate initialization, and the object will have a different value compared with its initial state. This is potentially dangerous to normal boot up.

Fig.6 is a good example to illustrate this problem. The booting codes are the same in every boot up; however, the initial value of global objects in the second boot up is different from the first one. The problem comes from the object's persistency, and because of this reason, it is possible that the object malfunctions if it is used in any other part of OS. For example, if a kernel spin lock object is initialized abnormally, the second boot-up process can fail as the spin lock blocks the process. However, it requires too much work to re-initialize every object, and what's more, some of the objects do not even need the process. Practically, it is impossible to re-initialize all objects in the kernel. Therefore, to reduce the amount of the work, the objects must be filtered before the initialization. This could necessitate an analysis of the kernel structure and the object type.

The kernel objects can be classified into three groups: (i) *logical objects*, (ii) *initialized global objects*, (iii) *uninitialized global objects*. The *logical objects* are assigned at temporary memory spaces such as stack and heap for space efficiency. It is unnecessary to re-initialize them because their initialization codes are in the texttt.text section; they are executed at the



First Booting	Second Booting

int i=7;
 	
int example(void)
{
    i=i+1;
}
	int i=7;
	
	int example(void)
	{
	    i=i+1;
	}

Fig. 6. The problem of persistent object

run time, and the codes are not changed during the run time. The *uninitialized global objects* are grouped into the `.bss` section. The section does not take up in the binary file, and it is initialized to zero when the OS boots up by initialization code in the *head.S*. The initial state of the objects in this section is zero, and it is not required to initialize them because they are initialized automatically in every boot up. The code in Fig. 7 is the section initialization code in linux 2.4.24.

```
not_relocated:      mov     r0, #0
1:                  str     r0, [r2], #4        @ clear bss
                    str     r0, [r2], #4
                    str     r0, [r2], #4
                    str     r0, [r2], #4
                    cmp     r2, r3
                    blo     1b
```

Fig. 7. .bss section initialzation

The *initialized global objects* are grouped into the `.data` section. In contrast with other objects, they must be re-initialized after the kernel shuts down for the next boot-up. They are initialized when the kernel is compiled, and their initial values occupy the `.data` section of the binary file. Owing to this reason, they cannot be initialized properly if the kernel loading is removed.

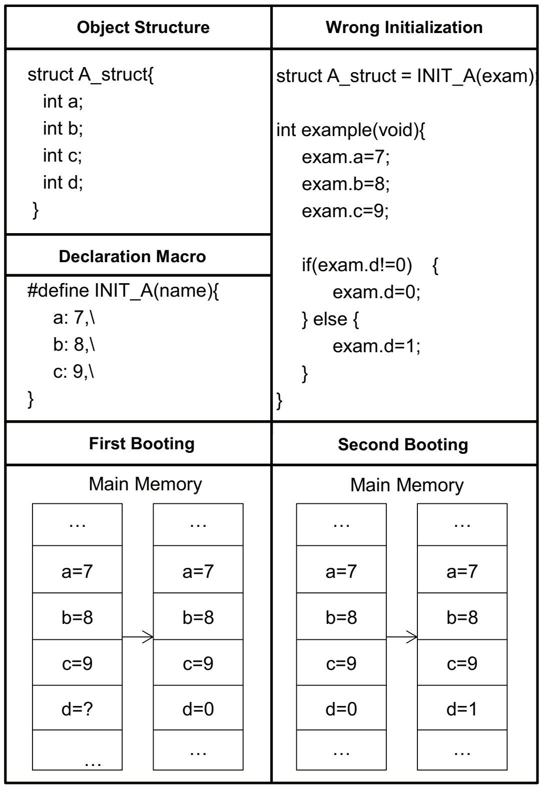
Our object filter extracts these objects from the kernel binary file and passes the information to explicit initialization module.

B. Explicit initialization module

The explicit initialization module consists of (i) *initialization code generator* which generates object initialization code semi-automatically, (ii) *initialization code inserting script*. There are hundreds of objects which is needed to re-initialize even if the object filter filters these objects out of a kernel binary. Therefore, the initialization code generator receives the information of the objects from the object filter, it generates the initialization code semi-automatically.

The simplest way to re-initialize is to change the values of objects to initial state. Based on this fact, the code generator makes a simple initialization code which restores the objects to the initial state. However, some objects are not completely initialized even if the initial state is restored. They have initial value of some members and the others are initialized when the kernel runs. Their value remains in the memory until next boot-up, causing an error when the OS boots up.

Fig.8 indicates the wrong initialization of an initialized global object. *'exam'* is a global object which is produced as '*A_struct*' form, and it has four int member variables. Member variable a, b, and c are initialized to 7, 8, 9, preceded by the initialization by '*INIT_A*' macro. The member variable *'d'* is initialized to zero during boot-up, but it has a different initial value at the next boot-up because the code flows differently. The *initialization code generator* initializes these members to zero by using a '`memset()`' function to prevent the error. The *initialization code inserting script* inserts the initialization code at the beginning of the kernel to initialize before they are used.



Object Structure	Wrong Initialization

Object Structure:

```
struct A_struct{
   int a;
   int b;
   int c;
   int d;
}
```

Declaration Macro:

```
#define INIT_A(name){
     a: 7,\
     b: 8,\
     c: 9,\
}
```

Wrong Initialization:

```
struct A_struct = INIT_A(exam);

int example(void){
     exam.a=7;
     exam.b=8;
     exam.c=9;

     if(exam.d!=0)   {
          exam.d=0;
     } else {
          exam.d=1;
     }
}
```

Fig. 8. Wrong Initialization

C. Dependency Check

Every kernel code is connected with each other. Therefore, if we remove the loading works, it is likely that the booting will be failed. In order to boot up stably with the absence of the works, we have to consider the effects of the removal. We have checked the dependence of the kernel image loading code and the ramdisk image loading code so as to successfully serve the purpose, and by this, we figured out that the works operate independently. Thus, it became evident that it is possible to safely remove the works.

V. Implementation

A. Bootless Boot Materials

To remove the kernel image and the ramdisk image decompressing, it is essential to make decompressed kernel and ramdisk image. Decompressed ramdisk image is produced by *'unzip'* operation as it was compressed by *'zip'* operation. There is a kernel building operation to produce a decompressed form of binary kernel that we can gain the decompressed kernel image.

B. Removing of Kernel Image Loading & Decompressing

Generally, the bootloader is responsible for loading a kernel image from the secondary storage. The uboot bootloader used in this research has an operation that loads a kernel image to a particular address of the main memory. We allocated the kernel image to a predefined physical address by using this operation and after that, it became no longer necessary to load the kernel image. The kernel was started from the predefined physical address by using an uboot operation which executes program codes from a particular physical address of the main memory, and we made it automatic by modifying the start configuration of uboot, and via the process, we succeeded in eliminating the linux kernel loading works. To eliminate the kernel decompressing, a function of decompressing the kernel image should be disabled. In our inspection, this function is in a *misc.o*. We have discussed earlier, even if we disable it, there is no effect on the kernel; therefore, it is possible to operate the kernel with stability.

C. Removing of Ramdisk Image Loading

On the whole, ramdisk image is allocated at a near address of a compressed kernel image. In Bootless Boot, it is feasible to damage to the ramdisk image because we use the decompressed kernel image. It can exceed the predefined ramdisk start address, so that we have to modify the start address of ramdisk, then we can remove the loading of ramdisk image in the same manner as the kernel image.

D. Keeping Initmem

Initmem is a memory space which has some kind of special objects that are only used during the boot-up time. After the boot-up, linux OS clears the memory space to manage the system resource efficiently. Since we use only one kernel binary, we will preserve the memory space for the next boot-up. In legacy linux, the removing process is comprised of one function which can help keep the *initmem* when disabled.

E. Object Filter

We have discussed earlier that global objects in the .data section have to be initialized. As far as it is concerned, it is important to find all of the objects in the section. Therefore, we tried to use the *'objdump'* to the kernel binary because it has all the information about the objects. We also found a section's start address and a section size from a kernel section header, and then calculated a section end address from this information. Next, we sorted the *'symtab'* (kernel symbol table) to find the objects in the `.data` section and filtered out the objects from the *symtab* by using the section start and the end address. Through the process, we could find the names and the sizes of the objects.

F. Explicit Initialization module

The biggest problem of initialization is that we have to spend too many times to generate the codes because there are hundreds of objects in the .data section. We made semi-automatic code generator to solve the problem. As previously discussed, every initialization code has *'memset()'* function inside. The *memset()* function needs only a name and a size of each object. This kind of information is already figured out from the object filter, so it is possible to generate *memset()* codes semi-automatically. We have used the text combining function. The information of *memset()* is used as an input, and the combining function assembles the text for code generation.

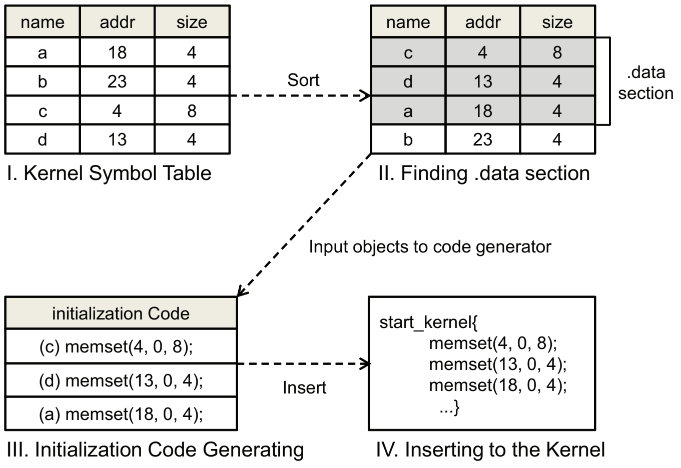


Fig. 9. Bootless Boot Implementation

Fig.9 is an instance to explain the code generation. The *initialization code inserting module* adds the generated codes to the kernel, and we changed the object's values to initial state for initialization.

	Kernel Image Copying	Ramdisk Image Copying	Kernel Image Decompressing	RootFs Mountaing	Initem Freeing	Bootless Boot Codes
Legacy Boot	1,432ms	3,792ms	748ms	4,372ms	4ms	0ms
Bootless Boot	0ms	0ms	0ms	1,051ms	0ms	32ms

TABLE II
GAIN AND LOSS ON APPLYING BOOTLESS BOOT

VI. EXPERIMENTS

A. Environment

We used linux 2.4.24 kernel which is optimized for MBA2440 embedded board, and conducted all our experiments on the board. The MBA2440 has the ARM920T CPU and 64MB of DRAM. Since the embedded board contains the DRAM main memory theoretically, it was impossible to test our technique on it. However, we have given a characteristic of non-volatility to the embedded board by a means of continuously supplying the power. We observed that the values of the initialized global objects were changed by booting number, and we could clearly prove that the assumption we made turned out to be correct. We connected the MBA2440 board with a host computer through uart and used the minicom[17] as a terminal program.

B. Boot-up Time Measure Mechanism

The general definition of boot-up refers to the time consumed from turning the power on to executing *'user init'*[7]. Therefore, we used an external timer to measure the time spent in the boot-up from the start of uboot to the start of user

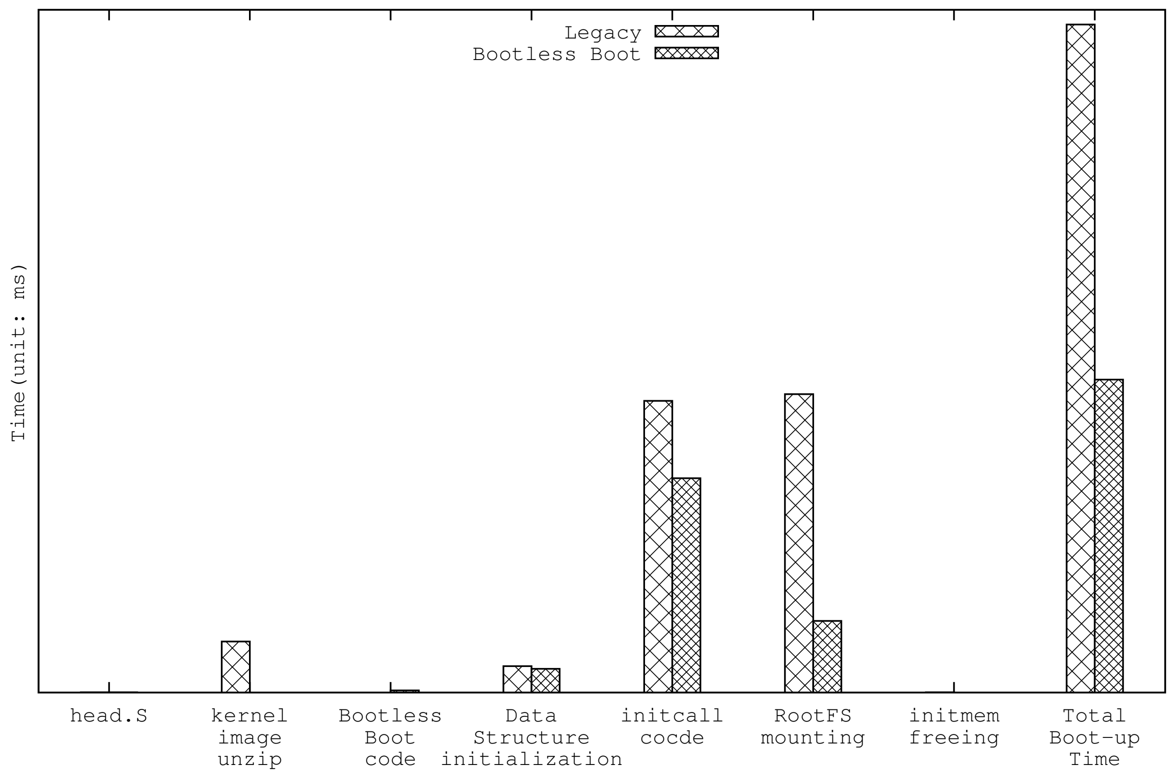


Fig. 10. Boot-up Time Comparison

Steps	head.S	Kernel Image Decompressing	Bootless Code	Data Structure Initialization	Initcall Code	RootFS Mounting	Initmem Freeing	Total Boot-up Time
Legacy Boot	4ms	748ms	0ms	388ms	4,272ms	4,372ms	4ms	9,784ms
Bootless Boot	5ms	0ms	32ms	348ms	3,140ms	1,051ms	0ms	4,584ms

TABLE III
COMPLETE TIME OF EACH STEPS IN BOOT-UP PROCESS

init in our experiment. When the minicom was operated on the host computer, the host timer started to work (tick=1ms), and then we made a simple function to print out a particular message to the host screen. We inserted this function at the beginning of uboot and at the start of kernel daemons so that the message could be printed out upon their starts. Also, we modified the minicom program to print out the timer's tick to the host screen if the message was transferred. We applied the function to measuring the total time of each work in the boot process, which turned out to be possible. As we have used an external timer, there marked a gap in time less than 3ms. It was just an increase of 0.019% in the entire boot-up time.

C. Gain and Loss on Applying Bootless Boot

Table.II shows the gain and loss on applying bootless boot. The original linux kernel image is 962KB and the ramdisk image is 2,179KB. In the legacy boot process, it takes 1,432ms to load a kernel image from NAND to DRAM, and 748ms to decompress. In the case of ramdisk, it takes 3,792ms to load a ramdisk image and 3,321ms to decompress. In Bootless Boot process, however, it takes 0ms to load and decompress each image because the process is unnecessary, and this is the greatest advantage of Bootless Boot. In fact, Bootless Boot kernel spends additional time of 32ms to execute the codes compared with the original one, but the amount is nothing more than a small portion, compared with the loading and decompressing time of images (9,297ms). In conclusion, we can save 9,265ms if we apply the Bootless Boot technique.

D. Boot-up Time Comparison

Fig.10 shows the complete time of each step in boot process between Bootless Boot kernel and legacy kernel. We measured all steps of the boot process and expected that it would show the same result, except loading and decompressing. The head.S step, the data structure initialization step, and the initcall step have no relations with the removed works. Therefore, they normally take almost the same time. However, the result which was obtained from the initcall step was opposite of what we have expected. We supposed that the result was affected by insertion of initalization code; however, we could not find the exact causes. Nevertheless, the boot-up was normally completed. Also, we observed that the boot-up time of legacy kernel is 9,784ms, whereas the boot-up time of Bootless Boot kernel is 4,584ms, leading to an increase of 213.4% in the speed of legacy kernel boot-up time. Furthermore, we should consider the effect of the removed works, which have saved 5,972ms, and this should be added to the time profit. Then, the boot-up time of legacy boot becomes 15,756ms. Therefore, Bootless Boot technique shows a high increase of 343.7% in the performance of legacy boot process.

VII. Conclusion

We have described Bootless Boot, a technique for reducing system boot-up latency on top of byte addressable NVRAM. Bootless Boot removes kernel and ramdisk loading as well as decompressing of these binary images. As a result, Bootless Boot reduces the boot-up time about 343% of the legacy linux kernel. The performance cost of the object initialization that Bootless Boot provides is very low, compared to the overall gains that Bootless Boot can offer.

Acknowledgment

New Memory: This work is supported by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for Newmemory based Smart Device).

References

[1] Kunhoon Baik, Saena Kim, Suchang Woo, and Jinhee Choi. Boosting up embedded linux device: experience on linux-based smartphone. In *Proceedings of the Linux Symposium*, 2010.

[2] Yi-jen Chen, Chi-sian Chuang, Yen-chung Chen, and Yun-tzuo Lai. Solid state drive, May 23 2013. US Patent 20,130,132,642.

[3] Kyung Ho Chung, Myung Sil Choi, and Kwang Seon Ahn. A study on the packaging for fast boot-up time in the embedded linux. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 89–94. IEEE, 2007.

[4] denx.de. U-boot [online]. 2010. URL: http://www.denx.de/wiki/U-boot/WebHome/ [cited Nov 2012].

[5] Swarnava Dey and Ranjan Dasgupta. Fast boot user experience using adaptive storage partitioning. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:*, pages 113–118. IEEE, 2009.

[6] SS Eaton, DB Butler, M Parris, D Wilson, and H McNeillie. A ferroelectric nonvolatile memory. In *Solid-State Circuits Conference, 1988. Digest of Technical Papers. ISSCC. 1988 IEEE International*, page 130. IEEE, 1988.

[7] Elinux.org. Boot-up time definition of terms [online]. 2012. URL: http://www.elinux.org/Boot-up_Time_Definition_Of_terms [cited Nov 2012].

[8] Miloš D. Ercegovac and Tomás Lang. *Digital systems and hardware/firmware algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 1985.

[9] Yiming Huai. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.

[10] Heeseung Jo, Hwanju Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Optimizing the startup time of embedded systems: A case study of digital tv. *Consumer Electronics, IEEE Transactions on*, 55(4):2242–2247, 2009.

[11] Heeseung Jo, Hwanju Kim, Hyun-Gul Roh, and Joonwon Lee. Improving the startup time of digital tv. *Consumer Electronics, IEEE Transactions on*, 55(2):721–727, 2009.

[12] Inwhee Joe and Sang Cheol Lee. Bootup time improvement for embedded linux using snapshot images created on boot time. In *Next Generation Information Technology (ICNIT), 2011 The 2nd International Conference on*, pages 193–196. IEEE, 2011.

[13] Li-Shi Ju, Yi-Chi Lai, Yueh-Min Huang, Yeni Ouyang, and Rory Tai. Fast boot methods of reducing customization: A case study of meego. 2010.

[14] Dongwook Kang. Enhanced ubi layer for fast boot-up times of mobile consumer devices. *Consumer Electronics, IEEE Transactions on*, 58(2):450–454, 2012.

[15] Shoji Koyama. Erasable, programmable read-only memory device, December 16 1986. US Patent 4,630,085.

[16] M.H. Kryder and C.S. Kim. After hard drives? what comes next? *IEEE Transactions on Magnetics*, 45(10):3406–3413, 2009.

[17] Adam Lackorzynski. minicom: Project home [online]. 2003. URL: http://alioth.debian.org/projects/minicom/ [cited Nov 2012].

[18] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[19] Alex Leonard. Boot time comparison: Iphone 4, moto atrix, galaxy s2 [online]. 2011. URL: http://alex.leonard.it/2011/06/18/boot-time-comparison-iphone-4-moto-atrix-galaxy-s2/ [cited Nov 2012].

[20] Panasonic. The new microcontrollers with on-chip non-volatile memory reram [online]. 2012. URL: http://panasonic.co.jp/corp/news/official.data/data.dir/jn120515-1/jn120515-1.html [cited 5 May 2012].

[21] Alberto Pesavento et al. Rfid tag using hybrid non-volatile memory, December 11 2007. US Patent 7,307,534.

[22] Ken Reneris. Computer hibernation implemented by a computer operating system, March 27 2001. US Patent 6,209,088.

[23] Jeff Tyson. How bios works. *How Stuff Works.[viitattu 19.10. 2012]. Saatavissa: http://computer. howstuffworks. com/bios. htm*, 2008.

[24] Xiaolin Wang, Zhenlin Wang, Shuang Liang, Zhengyi Zhang, Yingwei Luo, and Xiaoming Li. Fast booting many similar virtual machines. In *Systems and Virtualization Management. Standards and the Cloud*, pages 67–74. Springer, 2010.

[25] World Wide Words. Boot [online]. 2002. URL: http://www.worldwidewords.org/qa/qa-boo2.htm [cited Nov 2012].

[26] Kazuya Yamakita, Hiroshi Yamada, and Kenji Kono. Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 169–180. IEEE, 2011.

[27] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *FAST*, volume 5, pages 3–3, 2005.