Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 171 (2020) 1780–1789

Procedia Computer Science

www.elsevier.com/locate/procedia

Third International Conference on Computing and Network Communications (CoCoNet'19)

Implementation of Linux Optimization Technique for ARM Based System on Chip

Jasleen Kaur[a]* and SRN Reddy[b]

[a] PhD Scholar, Department of Computer Science, Indira Gandhi Delhi Technical University for Women, Delhi-110006, India
[b] Professor, Department of Computer Science, Indira Gandhi Delhi Technical University for Women, Delhi-110006, India

Abstract

Applications of an embedded system have been expanded throughout the years. It is usually observed in the digital consumer market where cell phones are replaced by smart phones and internet tablets. More functionality is brought into one single purpose system for giving the enhanced services to the users. The Linux Kernel based operating system is used in most of these electronic devices. The biggest protests about Linux are the size and speed at which it boots. The users expect these devices to have fast boot up time and a small amount of memory. In multiple real-time systems, many Linux versions are used that need a large level of accessibility and negligible downtime when updating systems. This results in advancing Linux boot time and size optimization. The objective of this paper is to keep the researchers up to date about Linux Optimization techniques. First of all, this paper survey the various boot time optimization techniques and size reduction techniques with respect to conventional Linux system. It then implements the efficient size optimization technique for ARM based Raspberry Pi board. The findings reveal that the operating system image builds after size optimization technique is 25% lighter in size than original image and boot-up time diminishes by half.

Keywords: Linux operating system; embedded software; Linux optimization techniques; Linux size optimization; Raspberry Pi optimization; ARM Based Board.

* Corresponding author
E-mail address: jasleenkochar89@gmail.com

10.1016/j.procs.2020.04.191

1. Introduction

In the past, the dimension of a PC was quite big. Now it has been turned into a small technology that can be conveyed into your palm. In terms of volume and simplicity, advancements have been made in hardware to enable more functions. As a result, to support these developed hardware technology (e.g. PC, cell phones, music players, etc.), the software has also been changed to an application specific custom operating system. Every day, the growing demands of customers are delivered with a huge weight of cost savings and an effective contour resulting into more complicated processes. A productive software and hardware install is needed to decrease idle time and to enable a quick update. Because the redesign of the hardware is expensive, the significant concentration moved to a software update. Relying on an open source OS, such as Linux, is a better strategy [1-4]. Linux has a free distribution OS that is supported by various computer architectures, for example, Intel, Atmel, Texas, Power PC and ARM [5-6].

The Linux OS fulfill the demands in terms of cost, features and usability for different applications areas such as embedded applications, personal systems and real time systems etc. The Linux operating system accepts all these functions because of its customizable behavior [7]. In [8], The Linux has been considered as the world's biggest and most advanced system. The Linux system has been used by many industry developers including Google [9], Sony [10], Flipkart [10], IBM [11] and several others. Optimizing the booting process of Linux must not alter the normal features. Startup optimization should not affect the functionality of the system, but it helps the system to improve the startup process for a faster system update.

The paper is structured as follows: the brief overview of other related work is described in segment 2; Segment 3 details the different Linux optimization techniques; Segment 4 defines the implementation procedure followed by results in segment 5; Segment 6 sums up the conclusion followed by the future work.

2. Literature Survey

In this advanced field majority of people depends upon the computer systems. The clients expect speed and effectiveness among systems, which rely upon the type of the OS installed in the system. The work done by the researchers in the area of Linux optimization has been covered in this segment.

The use of link-time compression technique is suggested in [12]. This method takes the advantage of the a priori known, fixed runtime surroundings of numerous embedded systems. The experimental settings are based upon ARM platform and it has been found that suggested approaches can decrease the kernel memory footprint with more than 16%. At the time of implementation, even after specialization, it has been noticed that there are still many seemingly useless codes in the kernel. Hence, codec compression methods are also proposed in [12] to diminish the impression of these useless codes. This method minimizes the kernel memory size with more than 28% for ARM based platform. In [13], hybrid root file-system strategy for a quick boot-up time in the Linux kernel is introduced. It utilizes the root filesystem and raw-kernel image that combine with JFFS2 and CRAMFS.

In [14], the means towards a quick booting Linux kernel utilizing non-intrusive techniques are detailed. The programming structure used in this work supports various real-time programming algorithms and does not change the source code of the Linux kernel. As a consequence, non-intrusive boot-up methods along with the unchanged Linux kernel and the uncorrected real-time scheduling module guarantee reliability and predictability. In [15], numerous strategies for lessening the Linux kernel boot-up time are discussed. The Linux operating system could be started on embedded technology in less than 600 ms by utilizing a variety of techniques discussed in [15]. In [3], initial research on the evolution of the best-known open source framework: the core of Linux operating system is summarized. It has been observed that Linux has been developing at a linear rate for quite a long while. In this paper, the authors evaluate the improvements in the core level of the operating system as well as within the main subsystems.

In [16], Hallinan explained how different configuration techniques can reduce the boot up time of Linux. In [17], the snapshot boot method is used to improve the boot up time, stability and efficiency of an embedded Linux based

system. In [18], sequence of the calling applications is visited again to enhance the user's observed boot-up time. In [19], an approach with SPM (Scratch-Pad Memory) is given in order to improve the boot-up time [19]. If the total boot up time cannot be reduced, hibernation is used to reduce the boot up time observed by the user. In [20], hibernation method to use the high response time available in the ad hoc mobile networks is discussed and in [21], hibernation method for WSN is discussed.

3. Linux Optimization Techniques

Enhancement is the way toward adjusting a software system to make some part of its work all the more proficiently or utilize less resources. Optimizing the Linux piece can enhance the system execution. Various optimization techniques used in multiple phases of the Linux booting process are described in segment. As displayed in Figure 1, optimization of Linux is widely split into two fields: Optimization of boot time and Optimization of size.

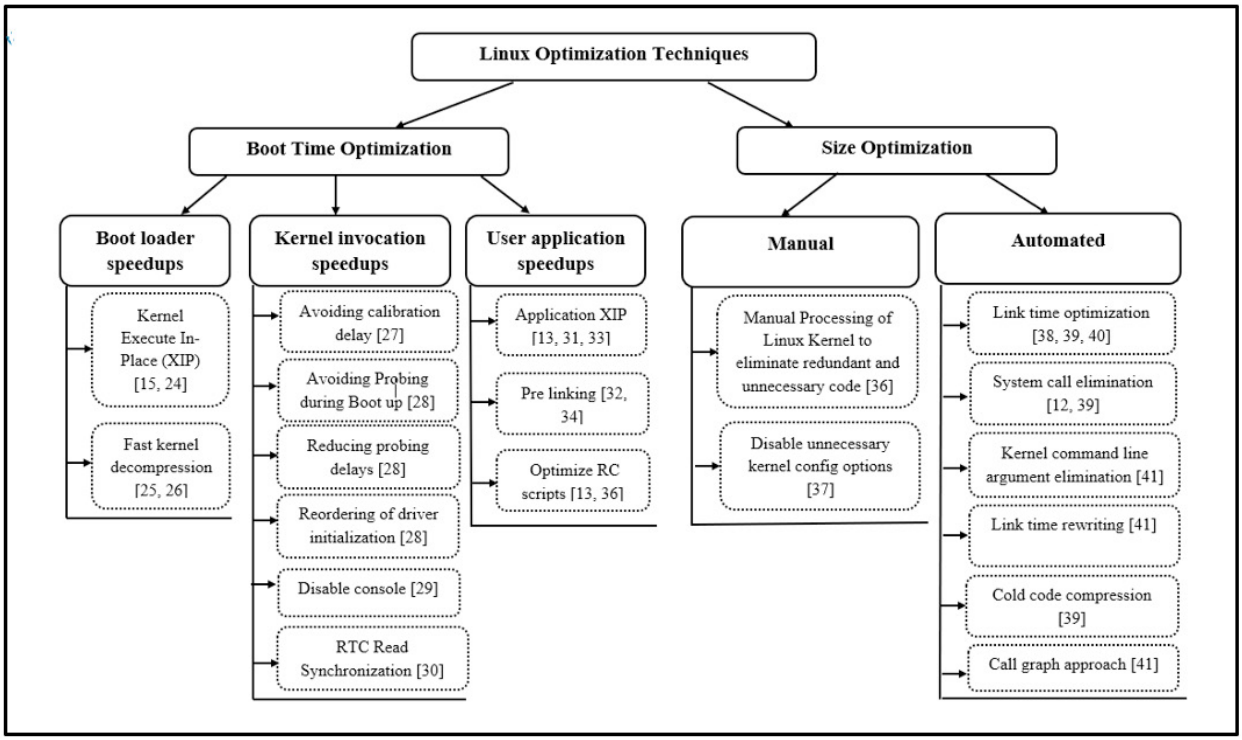

Fig. 1. Linux Optimization Techniques

3.1. Boot Time Optimization

It reduces the quantity of steps for booting (Optimize the Booting Time) and removes unnecessary highlights that are not required. There are various steps in the procedure to boot a machine running Linux OS. The boot time is the time, the system takes to make it usable as soon as the power button is pushed down. The specific elements, like boot loader, hardware and software configuration and Linux kernel settings, effects the boot time. Different boot time optimization methods which can be connected in various stages of booting process are discussed in this section. These methods can be utilized as a part of upgrading Linux in various stages of booting, for example, boot loader, kernel loading, client space and applications initialization. For a clearer image of the system, a detailed overview of the booting procedure is displayed in Figure 2. The optimization techniques for boot time are used for optimizing the Linux in various boot steps [15], such, boot loader, Kernel stacking, User-space and application initialization. In the following part distinctive optimization techniques that could be linked in various levels of the boot method has been discussed [22-23].

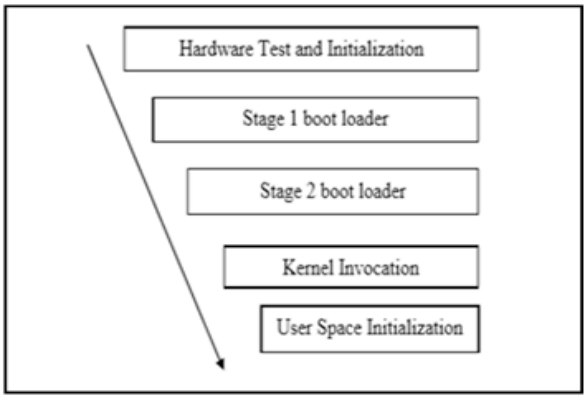

Fig. 2. Specific breakup of boot method [14]

3.1.1 Boot Loader Speedups

a) Kernel Execute In-Place(XIP)

The kernel image comprises different sections that includes: kernel data and code. The considerable amount of delay has been observed during decompression of Linux portion code. This delay can be precluded by the method known as Kernel Execute In – Place in which the kernel code portions can be placed in the flash memory in uncompressed frame and executed specifically from the flash. Examination consequences of this strategy appear in Table 1.

Table 1. Results of XIP and Non XIP [15, 24]

Boot Stage	Time with non XIP (ms)	Time with XIP (ms)
Copy kernel to RAM	85	12
Decompress Kernel	453	0
Kernel Initialization	819	882
Total kernel Boot Time	1357	894

b) Fast Kernel Decompression

In this method, advanced decompression algorithms are utilized as a part of request to lessen decompression time during booting. UCL is the compression methods focused by major embedded frameworks and provides the greater compression than gzip. Experimental results of utilizing this method in [15] infers that UCL decompression on Kernel image is 28% faster that gzip [27-28].

3.1.2 Kernel Invocation Speedups

a) Disable Console

While embedded architectures start with a serial terminal, the speed at which the characters are printed on the terminal is calculated by the serial yield speed, which can take a long time for basic processors. Boot time can be enhanced by disabling console yield [29] which can be accomplished by executing the "quiet" choice in the command line interpreter. This system will suppress the yield of general printk messages, yet these messages are placed inside printk buffer of kernel that could be recovered following the boot up method. Analysis consequences of utilizing this technique appear in Table 2.

Table 2. Results of Disable Console Technique [29]

Platform	Speed	Time without Quiet	Time with Quiet	Time Difference
OMAP 1510 (ARM 9)	168 MHz	551 ms	280 ms	271 ms

b) Avoiding Calibration Delay

Calibrate_delay () is the first routine in the Linux kernel that can be utilized to determine a loops_per_jiffy (LPJ) variable's correct estimation by implementing a delay progression. This method has been utilized to eliminate Linux kernel calibration by editing the code of caliberate_delay () placed inside init / main.c. The results of experiment with this technique by Noboru Wakabayashi [30] have saved about 212 ms time by executing OMAP TI (ARM) experiments with kernel version 2.4.20 [15, 27].

c) Avoiding Probing During Boot-up

This method is being used to prevent probing while booting. It consists in identifying the hardware that should not be present in the machine [9]. Add following arguments to ide.txt file in Linux/documentation directory: hdc=none hdf=none ide=noprobe. This method has been accessed since the 2.4-kernel series and founds to be a simple option to reduce the OS boot up time. The results of the No-probe experiment in [15] indicate that the complete saving time was 1700 ms.

d) Reconfiguring the driver initialization

The device driver is an application software that interact between the system and the processor and should be loaded during the boot-up process. There are two ways to stack the device driver in the kernel: dynamic and static loading. This method is utilized to enhance the loading of drivers. The device driver must be executed as a static driver when a device is needed at run at startup. Similarly, the device driver must be executed as a dynamic element when a device is not mandated to run at startup [28].

e) RTC Read Synchronization

The get_cmos_time () function has been implemented to inspect the outer RTC estimate, which took a long time to start the kernel. The syncing is simple to eliminate in this function by eliminating the initial control structures of the function [30].

3.1.3 User Space and Application Speedups

a) Application XIP

This method is used to enhance the application's startup speed and is similar to the technique of the XIP kernel. In this method, the kernel ought to aggregate with a file system that supports both linear and uncompressed file storage. CRAMFS [15] is a file system that supports these functions. The experiment was conducted using ARM-based XIP implementation [31] for which the findings have been shown in Table 3.

Table 3. Invocation Results of Application XIP and Non XIP [13, 31]

Invocation	Time with non XIP	Time with XIP
First Time	3.20 sec	1.75 sec
Second Time	2.04 sec	1.77 sec

b) Pre-Linking

This method is used to shorten the initialization of the application [32]. The static link requires large memory to store kernel images because the kernel collates all the common libraries, while the dynamic link is a slow process. That's why pre-linking method enables a dynamic link solution and significantly decreases the start-up time [33]. The experimental results of this technique are presented in [32-34], with Philips reducing the initialization time of the application by 30%.

c) Optimize RC Scripts

This method is utilized to optimize RC scripts [13] and shortens a lot of start-up time. At the point when the Init procedure begins, it runs distinctive RC contents with a specific end goal to call multiple system services. All contents in Linux are executed utilizing shell. The shells give several methods and commands to enable these heavy programs

and therefore running RC scripts using shells, provides the performance compensation. There is a tool named "Busy Box" [35] which can be used to optimize the RC scripts by combining the smaller versions of shell commands and utilities. The experiment findings in [15] gives a decrease of around 8 seconds in Debian RC scripts using an OMM 1510 based on ARM [35].

3.2. Size Optimization

This optimization technique helps in removing unused segments that are not required for a particular and reduce the size of binaries for each segment loaded. The size of the operating system is a major issue in an embedded area. Optimizations are nonexclusive and effectively viable. Manual tuning of Linux kernel size doesn't scale as an individual engineer can't be mastered in such a large number of various choices. The Linux kernel is growing 10% by consistently from most recent 10 years. Kernel 2.6.12 has 4,700 kernel config choices and kernel variant 3.9 has 13,000 config choices. It is important to recognize utilized code/required code from unused code/ unnecessary code while optimizing the size of Linux kernel. There are a couple of strategies for finding and disposing of unused code in the Linux kernel.

In Linux system there are lot of features that are not required. In order to decrease the size of the system, it is required to differentiate used code from the unused code. In manual size reduction technique, there is a manual processing of Linux kernel to eliminate redundant and unnecessary code by disabling the unnecessary kernel config options [36, 37]. There are several options in Linux kernel to control the features it supports [38]. The kernel, has been compiled with a large number of components and functions. By carefully tuning the kernel options, many parts of the kernel can be eliminated and memory can be saved. There are different size optimization techniques for Linux that comprises Link-time optimization, Kernel constraint system and System call elimination [39-41].

4. Implementation

In the current work, the Linux Kernel is optimized for ARM based Raspberry Pi board based upon size optimization technique and fast kernel decompression technique. Results are obtained for customized Linux image in comparison with official Linux [42] image for Raspberry Pi 3 model B. Results are detailed for 2 cases: a) Linux image with desktop environment b) Linux image without desktop environment. The official Linux image for Raspberry Pi 3 without desktop environment is named as Raspbian Lite [43].

The Table 4 depicts the host and target configuration used for implementing the size optimization technique and fast kernel decompression technique. Raspberry Pi 3 Model B and Kernel 4.1y is the targeted platform and the Linux kernel version respectively used in this article. The original downloaded Linux source code have been modified based upon the requirements and the minimum required drivers are linked to create customize light weight operating system image adequate for the targeted platform. The following steps are followed for optimizing the Linux kernel based upon optimization techniques and cross compiling the resulting operating system image for the ARM based platform.

Table 4. Host and Target Configuration Used

Parameters	Host OS Configuration	Target Configuration
Version	Ubuntu 16.04.2	Raspberry Pi 3 Model B
Memory (RAM)	8 GB	1 GB
Processor	Intel Core i7-4510U CPU @ 2.00GHz	Quad Core 1.2 GHz Broadcom BCM2837
OS Type	64-Bit	64-Bit
Disk	98.6GB	-

4.1 Kernel source code and tool chain

In the current work, the kernel source for BCM2837 chip placed on Raspberry Pi is downloaded and for cross compiling environment, the tool chain with linaro patches is downloaded with the help the commands shown in Table 5.

Table 5. Commands to download Linux source code and tool chain

```
$sudo apt-get install git
$git clone https:://github.com/raspberrypi/linux.git
```

```
$ git clone https://github.com/raspberrypi/tools ~/tools

$echo PATH= \$PATH:~/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin>>~/.bashrc

source ~/.bashrc
```

4.2 Kernel Configuration

The Linux kernel is the most configurable and the users can change the default configuration to their necessities, for example, removing the drivers that are not required, disabling the features that are never executed or enabling the support for new device drivers. The Linux kernel configuration is done by the make "menuconfig" utility [47] or by manually modifying the .config file. There are three different modes [48] in the configuration file: 'y' defines that it builds the selected functionality directly into the kernel, 'n' defines that it leaves the selected functionality out of the kernel and 'm' defines that it builds the selected functionality as loadable module that can be loaded when required. In the current implementation work, the configuration file (bcm2709_defconfig) targeted for the ARM architecture has been modified by making the mode changes in the functionalities which are not required in the minimum base kernel. After making the required changes, the final configuration file for the target processor is created with the commands shown in Figure 3.

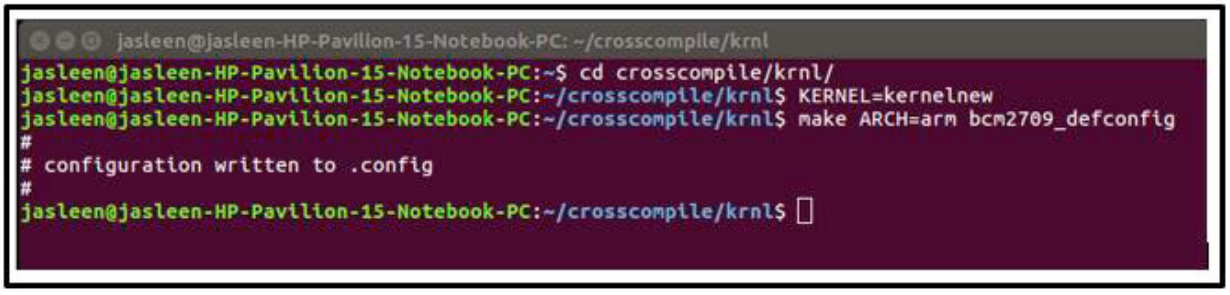

Fig 3. Creation of configuration file for targeted processor

4.3 Building and porting the kernel image on target processor

The building of kernel image through cross compilation process takes some hours and can be hasten by adding '-j<No. of CPU cores>". After the successful execution of cross compilation, the 'zImage' is build and placed in '/arch/arm/boot/zImage' folder. The zImage needs to be change to the binary file that is compatible with target processor. The cross-compilation process is executed and zImage is changed to the Raspberry Pi compatible 'kernel.img' image with the command shown in Figure 4.

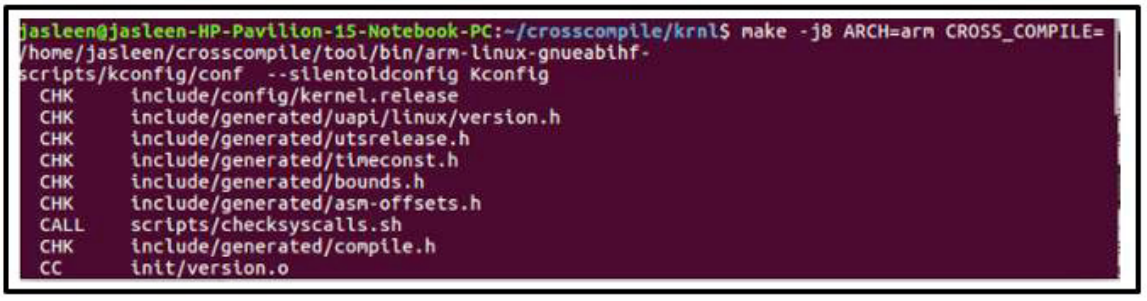

Fig 4. Execution of cross-compilation process and creation of zImage.

5. Results

As shown in Table 6, three distinct compression methods are used to compress the kernel image of 8 MB and 32 MB. Finally, the compressed kernel image has been utilized to start the operating system and time taken to compress the kernel has been observed by the use of PrintK method. The customized Linux image optimize with size optimization technique is ported in the SD card and tested with target processor. As shown in Table 7, the customized Linux is optimized in terms of various parameters such as total image size, minimum card size, memory usage, total processes, total installed packages, boot up time, root file system, scheduler, and system response time. It has been observed that customize operating system image that builds after implementing the size optimization technique is 25% lighter in size and boot-up time reduces by half.

Table 6. Experimental Results of Fast Kernel Decompression Technique

Compression Method Used	Time take to decompress the kernel (s)	Compressed size of kernel (Mb)
	When original size of kernel is 8MB	
gzip	0.252	4.1
bzip2	2.288	3.0
lzop	0.148	5.4
	When original size of kernel is 32MB	
gzip	0.892	15.8
bzip2	5.401	11.1
lzop	0.258	21.2

Table 7. Experimental Results of Size Optimization Technique

General State & System Optimization	Description	Without Desktop Environment	
		Original Image	Customize Image
Total Image Size	Complete size of the image. Smaller is preferred	1.49 GB	1.02 GB
Minimum Card Size	The smallest SD card size needed to run OS. Smaller is preferred	2 GB	2 GB
Memory Usage (RAM)	The complete memory used after booting. Smaller is preferred	27 MB	23 MB
Total Processes	The total number of programs and services executing after start up. Smaller is preferred	18	11
Total Installed Packages	The total number of installed packages. Smaller is preferred	406	265
Boot-Up Time in seconds	The total time, the system will take to show the prompt for login. Smaller is preferred	15.5 sec	14.2 sec
Root Filesystem Usage	Total usage of root filesystem. Smaller is preferred	931 MB	600 MB
Scheduler	Noop scheduler improves the system efficiency with SSD disks.	Deadline	Noop
System Response Times	Total time it takes to respond to a request for a service. Smaller gives good response time for the system	100 ms	25 ms

6. Conclusion & Future Work

In this paper various Linux optimization techniques have been discussed to make the system fast. The efficient Linux optimization techniques can be used for building lightweight application specific customize operating system for smart devices. Hence, in this paper, the research analysis on Linux optimization published by different authors have been presented. The paper briefly describes two areas for advancement in Linux, namely boot time optimization and size optimization. As discussed in this article, currently Linux kernel has been optimized by using size optimization technique and the operating system image build is 25% lighter in size and boot up time reduces considerably. The Broadcom BCM2837 based Raspberry Pi 3 and 4.1y kernel of Linux is used. The different advantages have been observed by using the above discussed Linux optimization techniques in Linux Kernel 4.1y that comprises: saving useless CPU idle time; saves startup time; increase system performance. The various potential hazards observed are: processor speed; error messages can influence system's stability; particular application performance consistently gets shortened.

Future research will concentrate on 1) experimenting these techniques with demonstrated instances of end user cases, for example building the minimal Linux image as per application specific requirement of the user; 2) experimenting these techniques on more ARM based hardware platforms such as ARM based Beagle Board and compare the results.

Acknowledgement

This work is supported by Microsoft University Relations, Finland under the research grant of project Mobile Education Kit-2 (Mek2) to Indira Gandhi Delhi Technical University for women, Delhi.

References

[1] J Sanders, "Linux, open source, and software's future", IEEE Software, 1998, Vol. 15, Issue 5, pp. 88-91.

[2] J. Hallen, A. Hammarqvist, F. Juhlin, "Linux in the workplace", IEEE Software, 1999, Vol. 16, Issue 1, pp. 52-57.

[3] Godfrey, Qiang Tu, "Evolution in open source software: a case study", Proceedings 2000 International Conference on Software Maintenance, 2000, pp. 131-142.

[4] N. Kshetri, "Economics of Linux adoption in developing countries", IEEE Software, 2004, Vol. 21, Issue 1, pp. 74-81.

[5] G. Unel et al., "Using Linux PCs in DAQ applications", IEEE Transactions on Nuclear Science, 2000, Vol. 47, Issue 2, pp. 109-113.

[6] S.R. Schach et al., "Maintainability of the Linux kernel", IEE Proceedings – Software, 2002, Vol. 149, Issue 1, pp. 18-23.

[7] Tatsuo Nakajima et al., "Issues for Making Linux Predictable", SAINT-W '02 Proceedings of the 2002 Symposium on Applications and the Internet (SAINT) Workshops, 2002, pp. 8.

[8] M.M. Lehman et al., "Implications of evolution metrics on software maintenance", Proceedings. International Conference on Software Maintenance, 1998, pp. 208-217.

[9] UC Davis (2002) Who Uses Linux? [Online]. Available at: http://www.lugod.org/presentations/ca4h/who_uses.html. Accessed on 20 Feb 2019.

[10] Ibrahim F. Haddad (2001) Open-Source Web Servers: Performance on a Carrier-Class Linux Platform [Online]. Available at: https://www.linuxjournal.com/article/4752. Accessed 25 Feb 2019.

[11] Linas Vepstas (2000) Linux on the IBM ESA/390 Mainframe Architecture [Online]. Available at: http://linas.org/linux/i370/i370.html. Accessed on 20 Feb 2019.

[12] Dominique Chanet et al., "Automated reduction of the memory footprint of the Linux kernel" ACM Transactions on Embedded Computing Systems (TECS), 2007, Vol. 6, Issue 4, Article 23.

[13] Kyung Ho Chung et al., "A Study on the Packaging for Fast Boot-up Time in the Embedded Linux", 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007, pp. 89-94.

[14] Mikael Asberg et al., "Fast Linux bootup using non-intrusive methods for predictable industrial embedded systems", IEEE 18th Conference on Emerging Technologies & Factory Automation, 2013, pp. 1-8.

[15] Tim R. Bird, "Methods to Improve Boot up Time in Linux", Linux Symposium, 2004, Vol. 1, pp. 79-88.

[16] C. Hallinan, "Reducing Boot Time in Embedded Linux Systems", Linux Journal, no. 188, 2009.

[17] I. Joe et al., "Bootup Time Improvement for Embedded linux Using Snapshot Images Created on Boot Time", ICNIT'11, 2011.

[18] Ubuntu Project, "The Ubuntu Project web page" [Online]. Available at: http://ubuntu.com/, 2004.
[19] N. Zhang, J. Ma, J. Chen, and T. Chen, "SPM-based boot loader," in ICESSSYMPOSIA '08: International Conference on Embedded Software and Systems Symposia, IEEE Computer Society, 2008, pp. 164–168.
[20] M. Hamdy and B. Konig-Ries, "Effects of different hibernation behaviors on the service distribution protocol for mobile networks and its replica placement process," in MDM '09: Tenth International Conference on Mobile Data Management: Systems, Services and Middleware. Washington, DC, USA: IEEE Computer Society, 2009, pp. 560–567.
[21] C. Feng, J. Peng, and H. Qing, "Phased waking coverage scheme based on hibernation of redundant nodes for wireless sensor networks," in ISCSCT '08: International Symposium on Computer Science and Computational Technology. Washington, DC, USA: IEEE Computer Society,2008,pp.709–713.
[22] Boot Time (2015) [Online]. Available at: https://elinux.org/Boot_Time. Accessed 10 Mar 2019.
[23] Chris Hallinan (2009) Reducing Boot Time: Techniques for Fast Booting [Online]. Available at: https://www.nxp.com/files-static/training_pdf/VFTF09_MONTAVISTA_LINUXBOOT.pdf. Accessed 10 Mar 2019.
[24] Kernel XIP (2014) [Online]. Available at: http://elinux.org/Kernel_XIP. Accessed 15 Mar 2019.
[25] Fast Kernel Decompression (2011) [Online]. Available at: https://elinux.org/Fast_Kernel_Decompression. Accessed 22 Mar 2019.
[26] UCL (2004) [Online]. Available at: http://www.oberhumer.com/opensource/ucl/. Accessed 28 Mar 2019.
[27] Preset LPJ (2008) [Online]. Available at: https://elinux.org/Preset_LPJ. Accessed 05 Apr 2019.
[28] Boot-up Time Reduction (2010) [Online]. Available at: http://elinux.org/Boot-up_Time_Reduction_Howto. Accessed 06 Apr 2019.
[29] Disable Console (2008) [Online]. Available at: https://elinux.org/Disable_Console. Accessed 28 Mar 2019.
[30] RTC No Sync (2008) [Online]. Available at: http://elinux.org/RTC_No_Sync. Accessed 08 Apr 2019.
[31] DENX Software Engineering (2003) Configure Linux for XIP (Execution In Place) [Online]. Available at: http://www.denx.de/wiki/bin/view/DULG/ConfigureLinuxForXIP. Accessed 18 Apr 2019.
[32] Pre Linking (2015) [Online]. Available at: http://elinux.org/Pre_Linking. Accessed 20 Apr 2019.
[33] K. Kato et al., "Embedded linux technologies to develop mobile phones for the mainstream market", 3rd IEEE Consumer Communications and Networking Conference, 2006, pp. 1073-1077.
[34] Changhee Jung et al., "Performance characterization of prelinking and preloading for embedded systems" EMSOFT '07 Proceedings of the 7th ACM & IEEE international conference on Embedded software, 2007, pp. 213-220.
[35] Optimize RC Scripts (2015) [Online]. Available at: http://elinux.org/Optimize_RC_Scripts. Accessed 25 Apr 2019.
[36] Chris Hallinan (2009) Reducing Boot Time: Techniques for Fast Booting [Online]. Available at: https://www.nxp.com/files-static/training_pdf/VFTF09_MONTAVISTA_LINUXBOOT.pdf. Accessed 25 Apr 2019.
[37] Kernel Configuration [Online]. Available at: http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/kconfig.html. Accessed 25 Apr 2019.
[38] Benjamin Schwarz et al., "PLTO: A link-time optimizer for the Intel IA-32 architecture", Proc. 2001 Workshop on Binary Rewriting, 2001.
[39] Tim Bird (2013) Advanced Size Optimization of the Linux Kernel [Online] https://events.static.linuxfound.org/sites/events/files/lcjp13_bird.pdf. Accessed 28 Apr 2019.
[40] Nicolas Pitre (2015) Linux Kernel Size Reduction [Online] https://linuxplumbersconf.org/2015/ocw/system/presentations/3369/original/slides.html#(11). Accessed 02 May 2019.
[41] Chi-Tai Lee et al., "An Application-Oriented Linux Kernel Customization for Embedded Systems", Journal of Information Science and Engineering, 2004, Vol. 20, pp. 1093-1107.
[42] Raspberry Pi official Image, "Raspbian" [Online]. Available at: https://www.raspberrypi.org/downloads/. Accessed 02 May 2019.
[43] Raspbian Lite [Online]. Available at: https://www.raspberrypi.org/downloads/raspbian/. Accessed on 04 May 2019.
[44] Sridhar Dhanapalan (2008) Introduction to Linux, Free Software and Open Source [Online]. Available at: https://linux.org.au/introduction-linux-free-software-and-open-source. Accessed 27 Feb 2019.
[45] IBM developer works (2007) Parallelize applications for faster Linux booting [Online]. Available at: https://www.linux.com/news/parallelize-application-faster-linux-booting. Accessed 04 Mar 2019.
[46] M. Tim Jones (2006) Inside the Linux boot process [Online]. Available at: https://www.ibm.com/developerworks/library/l-linuxboot/index.html. Accessed 08 Mar 2019.
[47] Configuring the kernel [Online]. Available at : https://www.raspberrypi.org/documentation/linux/kernel/configuring.md. Accessed on 04 May 2019.
[48] Configuring and Building [Online]. Available at: https://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/ch05.html . Accessed on 08 May 2019.