-

Analysis of Boot Time Reduction techniques in Linux

By
Abhishek Palod &
Rishabh Gupta

A Thesis Submitted to the
Faculty of Mälardalen University
In Partial Fulfillment of the
Requirements for the degree of

MASTER OF SCIENCE
Major Subject: Software Engineering

Mälardalen University
Västerås, Sweden
1st June, 2009

Daniel Flemström, Thesis Supervisor

Mikael Sjödin, Thesis Examiner

© Copyright 2009
By
Abhishek & Rishabh
All Rights Reserved

CONTENTS

LIST OF TABLES 6

LIST OF FIGURES 7

ACKNOWLEDGMENT 8

ABSTRACT 9

Svenska Sammanfattning 10

Acronyms 11

1. Introduction 12
   - 1.1 Structure of Thesis 13
2. Linux Booting Process 15
   - 2.1 Hardware test and initialization 16
   - 2.2 Stage 1 Boot loader 17
   - 2.3 Stage 2 Boot loader 18
   - 2.4 Kernel Invocation 18
   - 2.5 User space Initialization 19
3. Thesis Research Methodology 22
   - 3.1 Research Questions: 22
   - 3.2 Source of Literature 22
   - 3.3 Search Rule 23
   - 3.4 Search Drawback 23
   - 3.5 Process of Data Analysis and Synthesis 23
4. Boot Time Reduction Techniques 25
   - 4.1 Boot loader speedups 25
     - 4.1.1 Kernel Execute In-Place (XIP) 25
     - 4.1.2 DMA Copy of Kernel on Startup 27
     - 4.1.3 Fast Kernel Decompression 28
   - 4.2 Kernel invocation speedups 29

4.2.1 Disable Console 29

4.2.2 Avoiding Calibration Delay 30

4.2.3 Reordering of driver initialization 30

4.3 User-space and application speedups 31

4.3.1 Application XIP 31

4.3.2 Pre Linking 32

4.3.3 Optimize RC Scripts 33

5. Boot time measuring Tools 34

5.1 System wide time Measurement 35

5.1.1 Grab serial 35

5.1.2 Uptime 36

5.2 Kernel invocation time Measurement 36

5.2.1 KFT (Kernel Function Time) 36

5.2.2 Printk 37

5.3 User space time measurement 38

5.3.1 Boot Chart 38

5.3.2 Strace 39

6. Experiments 41

6.1 Experiment setup and goals 41

6.2 Results 41

6.3 Discussion 43

7. Conclusion 44

8. Future Work 47

8.1 Dynamic Update 47

8.2 Kexec 47

9. Appendices 49

9.1 POST process of an Award PC-BIOS [54] 49

9.2 Literature Search Result 50
9.3 Search Strategy [39] 52
9.4 Literature filtering Process 53
9.5 Sample Result of Grab Serial 54
9.6 Sample Result of KFT 54
9.7 Sample Result of Printk 55
9.8 Sample Result of Boot Chart 56
9.9 Sample Result of Strace 56
9.10 Experiment Result of Compression Techniques of Kernel Size 86.1 MB 57
9.11 Experiment Result of Compression Techniques of Kernel Size 54 MB 58
9.12 Experiment Result of Compression Techniques of Kernel Size 32 MB 58
9.13 Experiment Result of Compression Techniques of Kernel Size 8 MB 59
10. References 60

LIST OF TABLES

Table 1 Overview of POST..17
Table 2 Excerpt from the inittab file [52] 19
Table 3 Runlevel for Fedora, Red Hat and SuSE [53]...20
Table 4 Experiment Result of XIP and Non XIP [8][15] 25
Table 5 Comparisons of Non-XIP and XIP performance..26
Table 6 Configuration of DMA Technique [1]..27
Table 7 Performance chart of Various Stage of Booting by DMA [16][8] 28
Table 8 Invocation Results of XIP and Non XIP [76]...32
Table 9 Area of Measurement and Applicable Techniques..34
Table 10 Experiment Result of Uptime 36
Table 11   Experiment Results of Mitsubishi [46]...37

LIST OF FIGURES

Figure 1 Major step involved in a booting process [52] 15
Figure 2 Details breakdown of booting process [86] 16
Figure 3 Simple Example of Linux booting process [56] 21
Figure 4 Boot Time chart of DMA [58] 27
Figure 5 Experiment Result of Fast Kernel Decompression Technqiues 42

ACKNOWLEDGMENT

We would like to thank our supervisor Daniel Flemström for guiding us throughout the thesis and correcting us and helping us whenever problem occurred during entire period. We would like to thanks our examiner Mikael Sjödin for providing us valuable suggestion in order to improve the thesis. We would also like to give a big thanks to Mr. Lars olof Gustafson, coordinator of TCS Ericsson AIP program, by providing us the opportunity to undergo the master program at Mälardalen University in Sweden and also to our Company Tata Consultancy Services (India) which has given us the opportunity to undergo higher education in Sweden. We would also like to thank Mr. Par Asplund, Ericsson manager, who has guided us by providing valuable feedback regarding the subject.

And last but not least I would like to thank our TCS friends at Mälardalen University and all friends back in India for providing support and suggestions for completing our thesis successfully.

ABSTRACT

One of the biggest complaints about Linux is the speed at which it boots. Linux is a general purpose operating system which can be used as client or server right out of the box. Linux serves this wide base but it is suboptimal for any particular operation. Many variations of Linux are currently being used in various real time mission critical systems which require very high degree of availability and minimal downtime during system upgrades. This leads to optimization of boot time for Linux. In our thesis we will discuss applicability of different boot time reduction techniques with respect to traditional Linux systems and various measurement methodologies necessary to combine with reduction techniques in order to effectively optimize and measure the boot time. Finally we will present conclusions which can be used to improve boot time for a variety of systems running Linux.

The main focus of analysis will be the kernel and user space initialization of the boot process as there is little we can do about BIOS step. The important point to understand is that optimization of boot time should not compromise the system's existing functionality and stability by any degree but in turn help the system to enhance its booting process for faster system upgrade.

Svenska Sammanfattning

Acronyms

XIP	Kernel Execute In-Place
ROM	Random Access Memory
DRAM	Dynamic Random Access Memory
MS	Milliseconds
RAM	Random Access Memory
DMA	Dynamic Memory Access
CPP	Connectivity Packet Platform
OS	Operating System
CPU	Central Processing Unit
KFT	Kernel Function Time
GNU	GNU’s Not UNIX
PNG	Portable Network Graphics
SVG	Scalable Vector Graphics
SDRAM	Synchronous Dynamic Random Access Memory

1. Introduction

In mid 20th century, the size of a computer was almost as big as a room. Today it has become a device which can be carried in your hands. Evolution occurred in hardware in terms of size and complexity to support more features. Accordingly, software has also upgraded from traditional punch card to a customized operating system to support these advanced hardware devices (for example: PCs, mobile phones, music player etc). A telecommunication node running critical applications is built upon a very complex multi processor architecture and the platform used for this architecture can reach a high degree of complexity. Day by day increasing customer requirements are mounting enormous pressure of cost reduction and efficient design which results in more and more complex systems [1]. To reduce downtime and allow smooth upgrade we require efficient software and hardware upgrade. As hardware upgrade is costly [2], the major focus shifted to software upgrade. One common solution is to rely on a proprietary operating system. This has been proven to be very costly and time consuming [1]. A better approach is to rely on an open source operating system such as Linux [25][28][29] [30]. Linux has free distribution OS which runs on almost all architectures such as Intel, PPC and ARM with extension from application platform's for real time mission critical systems to standalone [24][27]desktop computer. Linux meets the requirements of everyone in all fields such as embedded, real time, personal computer in terms of functionality, scalability and cost. Linux supports all these features because of its configurable nature [1][22]. Linux [1][3] [25] is world‘s biggest and longest evolved software system. Linux is used by most of the market leaders product development and business such as IBM [4], Google[5], Amazon [5], Ericsson [6][7] and many more. Various mission critical systems such as an aircraft control system or an application platform running on a communication node require system to be started up in few seconds to a minute [8]. But delay of few seconds can cause disasters in terms of safety and business. In order to reduce the system downtime, it is necessary to examine the boot process of the operating system used which is Linux in this discussion. To do so, a large variety of questions need to be answered such as different factors affecting the booting process, major areas having significant scope of improvement etc. An analysis of the boot process is challenged by fast development by Linux community which results in more and more complex Linux systems.

Optimizing the Linux Bootup process should not override or change the existing functionality, nor the normal operation. On optimization we extend the system to a level where it reduced the time of the booting process, but fail to satisfy normal operation then whole process will be lost. Boot optimization should not affect the system functionality, but in turn help system to enhance its booting process for faster system upgrade.

1.1 Structure of Thesis

- Chapter 2
  This chapter covers the details of major steps involved during Linux booting process. The chapter presents a detailed explanation of working of Linux booting process steps such as stage 1 boot loader, stage 2 boot loader, kernel invocation and user space initialization.

- Chapter 3
  Chapter 3 discusses research methodology used during the thesis. In this chapter the research questions of thesis was highlighted. The steps involved during the process of literature collection was discussed which includes different scientific databases, search rule and the process of data analysis and synthesis. The drawback of search was also discussed in this chapter

- Chapter 4
  Chapter 4 cover detail explanation of different Reduction techniques. The chapter includes the applicability of different boot time reduction techniques for major Linux booting steps such as boot loader, kernel invocation and user space initialization was discusses. The advantages and drawbacks of each technique were discussed.

- Chapter 5
  This chapter explains and analyzes various boot time measurement methodologies. The different measuring techniques based on its applicability to Linux booting steps is discusses .The advantages and disadvantages of the techniques was also elaborated in this chapter.

- Chapter 6
  Chapter 6 focuses on experiments conducted, which is followed by analysis of its results. In this chapter the experiment results based on applicability of reduction technique is discussed. The result obtained from the techniques is discussed in the discussion section.

- Chapter 7
  This Chapter covers the conclusion drawn during the thesis work. The conclusion depicts the applicability and suitability of reduction techniques and measuring method was highlighted.

- Chapter 8

  Chapter 8 presents a scope of improvements and future work. The major techniques such as Dynamic update and Kexec was discussed. The major advantages and pitfalls of the techniques were discussed. The applicability and concerns of techniques with respect to Linux is also highlighted in this chapter.

  .

2. Linux Booting Process

The process of booting a system running Linux operating system consists of multiple stages. Much of the flow of this process is similar whether we are booting an x86 based personal computer or an embedded system based on PowerPC architecture. In this section, we will explain this process from the point of powering on or reset of the system to the first time when user sees the login screen. In this chapter, we will learn about different aspects of booting process such as Boot loader, kernel decompression and initial ram disk. Understanding of all these sections will serve as a basis for the rest of the discussion. The Entire process can be divided into three major parts as shown in Figure 1.

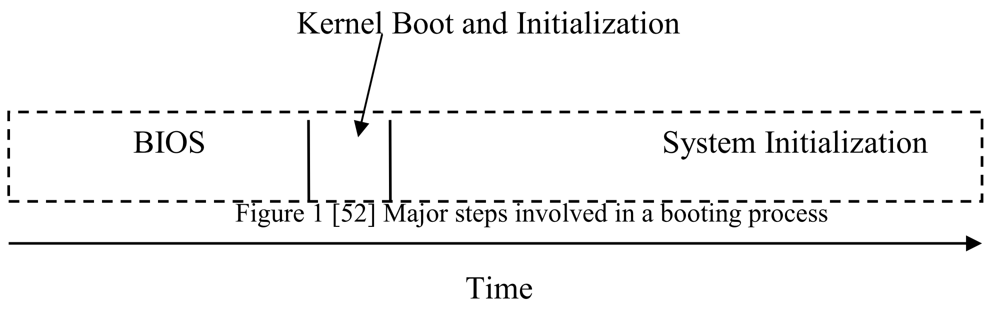

Kernel Boot and Initialization
BIOS
System Initialization
Figure 1 [52] Major steps involved in a booting process
Time

Figure 1 Major steps involved in the booting process **[52]**

Figure 1 shows three major parts of the booting process on a time scale which means it provides as estimate of time taken by each step as a fraction of total boot time. In Figure 2 (shown below), we break down these steps and show another view in order to have a clearer picture of the process.

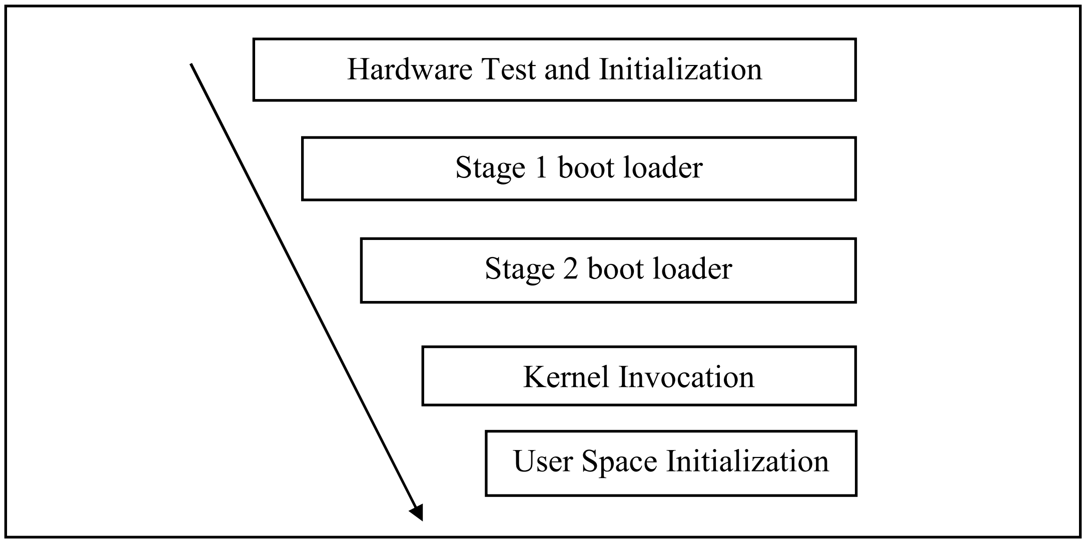


Figure 2 Details breakdown of booting process [86]

2.1 Hardware test and initialization

The hardware test and initialization stage is hardware dependent and differs for different architectures. An embedded platform uses a bootstrap program (for example: U-Boot, Red-Boot and Micro Monitor from Lucent). These programs shipped along with the embedded platforms and stay in a specific region of flash memory on the target hardware and provide an interface to download the Linux kernel image into the flash memory and execute it later. Also, these programs perform the hardware test and initialization and cover both the first stage and second stage Boot loaders [57].
On the other hand, in a PC, BIOS resides at address 0xFFFF0 and the first step of BIOS is Power On Self Test (POST). POST can vary from manufacture to manufacture (Asus, Mercury and Intel) and version to version, because there is no standard exists. An overview of steps involved in POST is given in table 1 below (confer appendix section 9.1 for details).

Table 1 Overview of POST

CPU Test
Memory test and initialization (RAM, Flash, cache)
I/O devices and their controllers initialization
Other peripheral devices and their controllers initialization
Test and initialize CMOS settings
Configuration settings for booting process (boot device priority, error handling, default core speed)

BIOS can be divided into two parts responsible for different functionalities, POST code and BIOS runtime services. Once the POST is completed, it is erased from the memory but runtime services stays in the memory because later they are used by operating system.
After POST, in order to boot the operating system, runtime services search for devices which are active and bootable in the order of preference set by CMOS settings which is user configurable. A boot device can be a floppy drive, a CD-ROM, hard disk, network device or even an usb stick.
Generally, Linux is booted from a hard disk partition where master boot record (MBR) contains the stage 1 boot loader. Stage 1 boot loader known as primary Bootloader is a 512 byte sector on the first sector of the hard disk. Once MBR is loaded into RAM, BIOS transfer control to it. [86]

2.2 Stage 1 Boot loader

The Primary boot loader which resides in MBR is a 512 byte image containing program code and a partition table. The First 446 bytes are the primary boot loader having executable code and the error message text. Next 64 bytes are partition table. The last two bytes of MBR are called the so magic number (0xAA55) and is used for the validation check of MBR.
The responsibility of the primary boot loader is to find and load the stage 2 boot loader (secondary boot loader). It performs this task by searching the partition table for an

active partition. On finding an active partition, the primary boot loader verifies that all other partitions in the partition table are in-active. After this verification,the boot record of active partition is loaded into RAM and starts execution [86].

2.3 Stage 2 Boot loader

The secondary Bootloader is also known as Kernel loader as its main functionality is to load the Kernel image and optional Initial ram disk (explained in detail later in the chapter). For x86 based systems,the combination of the first and second stage boot loaders are called Linux Loader (LILO) or Grand Unified Boot Loader (GRUB). The major difference between these two is that GRUB is a file system while boot loader and LILO are not (read subsection 2 for details at the end of this chapter). This file system awareness of GRUB provides greater advantage over LILO because instead of using raw sectors on the disk, GRUB can load kernel from an ext2 or ext3 file systems. It achieved this by dividing the two stage boot loader process into the 3 stage boot loader process. Stage 1 (MBR) loads a stage 1.5 boot loader which understands the particular file system containing the Linux kernel image. When the 1.5 stage boot loader is running, stage 2 boot loader can be loaded into RAM. This part of the process consumes a significant amount of time and provides ample scope for improvement.
After stage 2 is loaded, GRUB displays a screen showing all available operating systems. The user can select desirable kernel and can even give some additional kernel command line parameters. This is followed by loading and invoking of the kernel image into RAM [86].

2.4 Kernel Invocation

When control is given to the kernel loaded into the memory,the kernel stage begins. The kernel image is not executable but it is a compressed binary image. Typically this is compressed by gzip in traditional Linux systems. At the head of this kernel image, there is a small routine which performs decompression of the kernel contained in this image and put it into the high memory. If an initial ram disk (initrd) is present, this routine copies it into memory for later use. The routine then calls the kernel and the kernel boot begins.
During kernel boot, initrd, which is loaded into memory by stage 2 boot loader is now mounted. The initrd serves as a root file system in RAM and allows the kernel to boot fully without mounting any physical file system. The positive feature of initrd is that drivers needed by the kernel to have an interface with devices can be combined within initrd. Hence, a small kernel image is capable of supporting a large number of possible hardware configurations. After the kernel is booted, the actual room file system is pivoted, where initrd is unmounted and the real root file system is mounted.
Initrd allows us to create a small kernel with drivers compiled as loadable modules. These loadable modules provide the kernel interfaces to access disks and file system on these disks and also drivers for other hardware devices. As the root file system is a file system on disk, initrd provides a way to mount and access this file system. In case of an

embedded system without a hard disk, initrd can work as final root file system or a final root file system can be mounted via the network file system (NFS) [86].

2.5 User space Initialization

The final step of the booting process is explained in the following section written for Fedora (which is one of the most widely used Linux systems) and it may differ slightly for other flavors of Linux systems.

The first process of Linux is Init. Init is the parent of all processes in Linux system. After the kernel invocation, kernel starts Init which in turn is responsible for initialization of all other background processes and system services/

As soon as Init starts, it runs a /etc/rc/d/rc.sysinit script which performs specific functions in order to proceed with system initialization such as setting up execution paths, start swapping and checking of file systems etc. This script also uses the /etc/sysconfig/clock file to initialize the system clock.

Above steps are followed by execution of the /etc/inittab script. This is the configuration file for init which determines how the system should be initialized. This file also determines what to do in case of power failure and in case of some specific key sequences (for example ctrl+atl+delete) (confer Table 2 Excerpt from the inittab file)

The inittab file contains several entries having common format: **id: runlevels: action: process**. *The id is a sequence of characters that uniquely identifies the entry. The runlevels define the runlevels for which the action should be taken. The action specifies the particular action to take. Finally, process defines the process to be executed [86].*

Table 2 Excerpt from the inittab file [52]

```
# The default runlevel
id:2:initdefault

# Boot-time system configuration/initialization script
si::sysinit:/etc/init.d/rcS

# Runlevels
l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
```

```
l6:6:wait:/etc/init.d/rc 6
z6:6:respawn:/sbin/sulogin

# How to react to ctrl-alt-del
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
```

Table 3 Runlevel for Fedora, Red Hat and SuSE [53]

Runlevel	Description
0	Halt
1	Single user mode
2	Multi user without network, without NFS
3	Full multi user mode, without x-based login screen
4	Not used (User definable)
5	Full multi user mode with x-based login screen
6	Shutdown with reboot

After Init finishes Execution of /etc/inittab, it brings the system to its default runlevel which is runlevel 2 in this case (table 3).
Whenever the runlevel changes, Init starts and stops various services defined for that runlevel. The runlevel can be considered as state of the system. For example, runlevel 0 defines the system halt state; runlevel 1 is the single-user mode. Runlevels 2 through 5 are multi-user states, and runlevel 6 indicates reboot.
Once the default runlevel is set, Init starts all background processes which are required for the system to run. These processes are started by looking into the appropriate RC directory for their respective runlevels (/etc/rc.d/rc<x> where x is the runlevel).

After Init runs through all runlevels,the Inittab script invokes various different processes for login consoles and at this point the user should be able to see login prompt.

Figure 3 The booting process once the secondary boot loader is loaded.

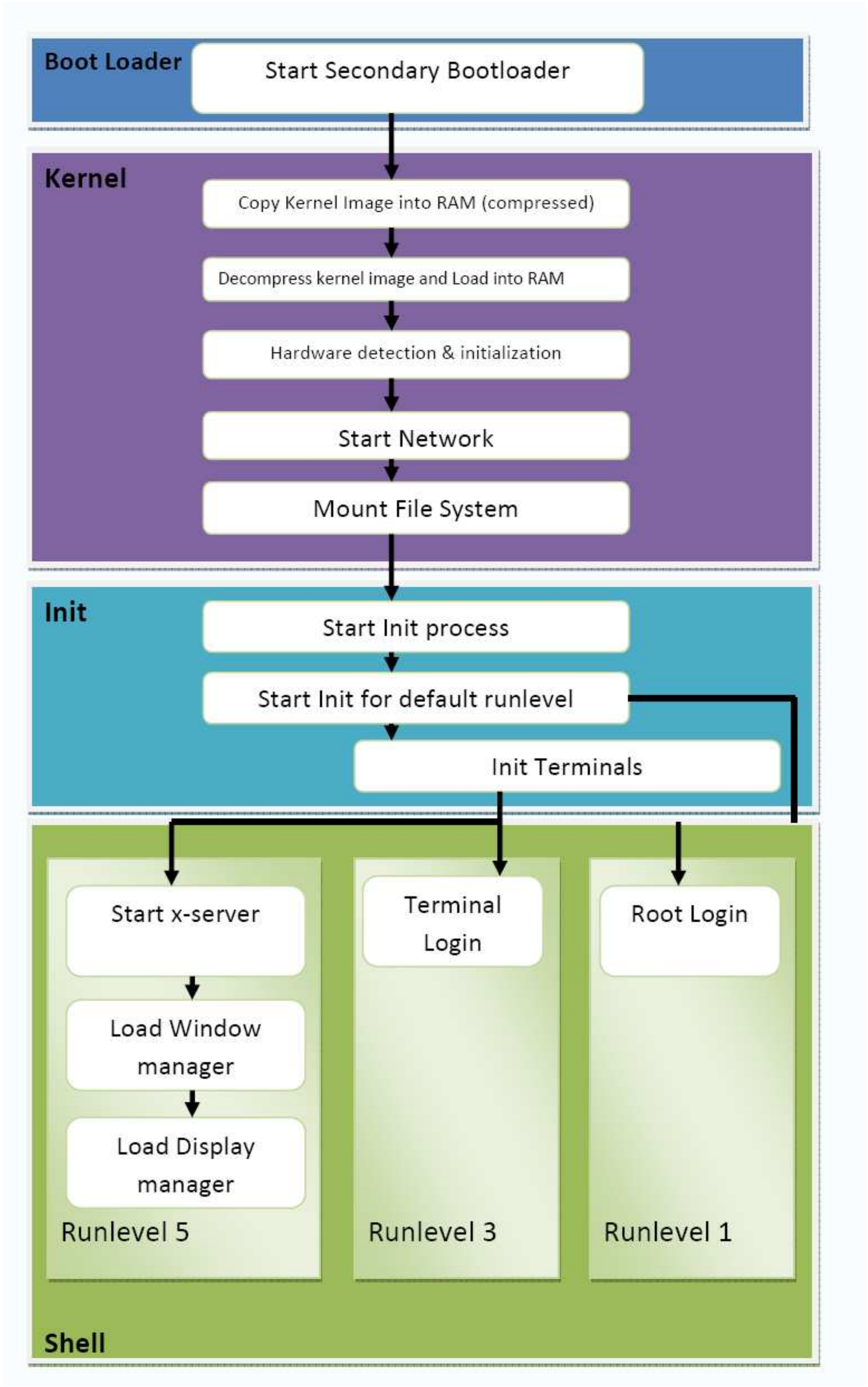


Figure 3 Simple Example of Linux booting process [56]

3. Thesis Research Methodology

The master's thesis is a challenging area for a student in terms of problem solving activities, project work and assignments. The thesis has no clear cut goals or any definite structure or processes to achieve the goals. Students are required to be active and practice self learning to define the problem statement. Its self learning where problem statement needs to be found and solutions need to be discovered [33].A Thesis is exploratory research mentioned in Cooper& Scheindler[34]. Macdaniel & Gates say" exploratory research is a small research which is taken to define exact nature of problem and to understand the nature of environment where problem occurred" [34]. Through which, we are able to learn not only science but also processes and methods [34].In our research we followed Kitchen's Evidence based software engineering principles (EBSE) [36]. By using EBSE and systematic review process [35][37] we were able to longitudinally study and explore different aspects of boot time reduction techniques of Linux.

3.1 Research Questions:

The thesis aims to find the causes and solution of slow boot time in Linux through research by formulating the following questions. [38]

1) What are the areas causing significant boot time delays?
2) Which different reduction techniques can be considered?
3) Can these techniques be classified according to the Linux booting process?
4) What are the pros and cons of these techniques and how are they measured?
5) Can one technique be showed better than the other, if yes then how?

3.2 Source of Literature

Kitchenham's guidelines [37] were followed to gain broader prospective and improve the quality of information, we searched different electronic mode of information. Below is the list of different databases we examined.

- IEEE
- ACM
- ScienceDirect
- Web of Science
- SpringerLink
- Wiley InterScience

According to Dybå [39] the databases listed above contain most recent journal and conference in software engineering. Apart from then above databases, Google scholar also proved to be efficient for searching web links since the majority of work in Linux is taking place in the developer community, there is a lack of published scientific papers hence we were unable to find relevant literature in scientific databases.

3.3 Search Rule

In a preliminary stage, search criteria was quite vast in order to include articles of different areas related to Linux. The search key words were Linux, Boot Time, Optimization and Kernel. The relevant literature was searched by using Boolean expression "OR" and "AND" operator. The search includes articles and web links sources till April 2009. By applying different search combinations sequentially, search criteria were reduced. The initial search located 51024 of most relevant articles excluding Google scholar search of 651000 articles. The different search results were shown in section 9.2.

3.4 Search Drawback

Different search criteria were needed to decide the different databases. For example in IEEE only key words based search works, on addition of Boolean operator results found are zero. Also, by using combinations of different search keywords together, no results were found. On the other hand, ScienceDirect and Wiley of Science display all records on basis of keywords but due to registration problem, we were not able to view all literature. SpringerLink displays all record when one word matched in its full text of literature, because of this strategy, irrelevant records were displayed. ACM comes up with different literature but in order to prevent copyright violation we required a license to access them. In case of Google scholar, all string combination works and displays a huge number of results that is why we needed to take only top searches for literature review.

3.5 Process of Data Analysis and Synthesis

The three ways of the selection process described by Haftmann [40] was applied to find suitable and relevant literature. In this process as described in 9.3, initial literature was identified in different databases, books, journal and websites. Elimination in next stage was achieved using filtering by titles and abstract. Further processing was carried out by systematic review of content and conclusions to get the most desired sources of information. The different search criteria results obtained from different databases are shown below in 9.2. The final sources after different levels of filtering were 10 as shown in 9.4. In this process of final selection a thorough review of literature was carried out by both team members on basis of reading content and checking relevancy on basis of questions obtained from problem statement. By this process we were able to narrow down our literature. Apart from this process, we came to the conclusion that there is a lack of scientific papers on boot optimization of Linux we need to rely on open sourced web links to get information.

4. Boot Time Reduction Techniques

A big question arises, how to reduce boot time without affecting performance and existing usability? Boot time is affected by different factors [11] such as hardware, boot loader configurations, kernel configuration and application profile. We will discuss various Boot reduction techniques which will be used for optimizating Linux in different steps of booting [8] such as boot loader, Kernel loading, User-space,application initialization and so on. We will discuss different boot reductions which can be applied in different steps of the booting process. [10][11].

4.1 Boot loader speedups

The boot loader speedup techniques are the ones which can be applied to stage 1 boot loader and stage 2 boot loader as discussed in chapter 3 while discussing the booting process. The techniques we are going to discuss in this section shows possibilities to reduce boot time in these steps and sets a platform for the future work in this area.

4.1.1 Kernel Execute In-Place (XIP)

In the traditional booting process, the kernel image (optionally with Initrd) is copied into RAM by a small routine which sits on the top of the image. During this process, the routine decompresses the image and copies the decompressed binary kernel file into RAM. The kernel image is nothing but a combination of two segments: kernel code and kernel data. Traditionally, a compressed kernel image is created by combining these two segments into single binary image. There is a significant amount of delay during decompression of the Linux kernel code and also initialization of kernel data segment in RAM [8].

Looking back to figure 3, the steps of copying the kernel image to RAM and decompressing it are omitted. Instead, kernel code segments are stored in the flash memory in uncompressed form. Only kernel data segments are copied into RAM and kernel code segments are executed directly from the flash. By doing this, time consumed in copying kernel code segments and decompression is saved [8][10]

An Experiment using XIP was conducted by Tim Bird (From Sony Electronics) [8] on Power PC 405 LP at 266 MHz and time duration of various parts of the booting process with and without use of XIP was measured. The result shows that the total time for copying the kernel to RAM is reduced (because only data segments were copied) and omitting the decompression step completely. The following table shows the comparison of time of normal booting compared to time of booting using XIP [8].

Table 4 Experiment Result of XIP and Non XIP [8][15]

Boot Stage	Non-XIP time duration(ms)	XIP time duration
Copy kernel to RAM	85	12
Decompress kernel	453	0

Kernel initialization	819	882
Total kernel boot time	1357	894

By analyzing this table, we found that there is a 75.89% reduction of time during kernel copy to RAM, 100% save during the decompression stage and an overall reduction of 34.11% in kernel copying and loading steps. In an environment using big kernel images, this time is significant.

However, few overheads was discovered from the experiment performed by Bird for XIP. Executing the kernel code from RAM is much faster than executing it from the Flash memory which in turn causes slower kernel execution when using XIP. An experiment result conducted with OMAP 1510 (ARM9 at 168 MHz) processor which is shown in table 6. It depicts the kernel performance comparison by used XIP and non XIP methods. [8] [14]

Table 5 Comparisons of Non-XIP and XIP performance

Operation	**Non-XIP time duration(ms)**	**XIP time duration**
Stat() syscall	22.4	25.6
Fork a process	4718	7106
context switching for 16 processes and 64k data size	932	1109
pipe communication	248	548

Table 5 depicts that major operations involve during kernel invocation. The major point we can observe is that there is a considerable performance drop while executing kernel code directly from flash because the flash memory is much slower than RAM. For example, this performance drop is 30% while invoking fork (system call). These drops cause overall performance delay.
The second major disadvantage is that the size of the flash needs to be increased, since storing an uncompressed kernel image requires 30%-50% (based on compression technique used) more space. This incurs extra cost of flash memory. [8] [14]

From the above discussion we can conclude that applicability of XIP largely depends on the hardware configuration of the system. For example, for an average size a kernel, a system having Flash memory constraints and fast CPU does not require XIP because it is not feasible to store an uncompressed kernel into a smaller flash memory and a very fast CPU makes compression time negligible (for example; Mobile phones or other small embedded devices). On the other hand, if the system consists of a large amount of flash memory and average speed CPU, then using XIP can be very useful (For example Personal Computer).

4.1.2 DMA Copy of Kernel on Startup

Direct Memory access (DMA) is an aspect of today's advanced computer systems. It allows hardware devices to read/write data to system memory without communicating with the CPU. This feature reduces CPU usage and gives better performance in comparison with the systems without DMA.
Jyunji (Kondo Fujitsu Software Technologies Limited) [16] conducted an experiment on Fujitsu FR-V Design Kit for FR450 core on Linux kernel version 2.6.6. The configuration settings made in this experiment are shown in Table 6 below.

Table 6 Configuration of DMA Technique [16]

Configuration Settings
DMA kernel copy - enabled/disabledBR
PresetLPJ - enabled/disabledBR
Avoiding printk messages - enabled/disabled

Experiment results are shown in Figure 4 which demonstrates comparison of normal booting process and booting process using DMA copy of kernel on Startup (shown as fast boot in figure). The major time consuming task is kernel initialization and user initialization as depicted from the Figure 4 figure below.

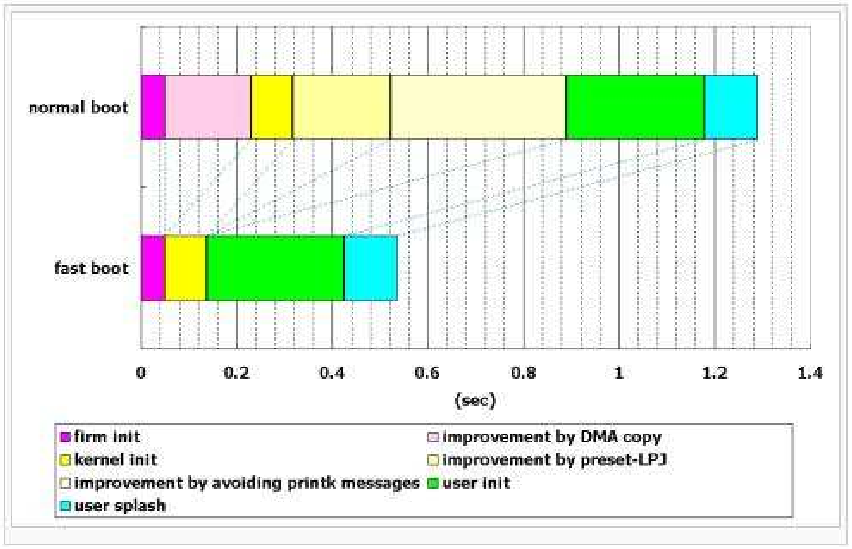


Figure 4 Boot Time chart of DMA [58]

During the process, CPU copies the kernel image of about 1.7 MB from Flash to RAM. This is part of the stage 2 boot loader step and total time consumed in this can be reduced by DMA [14] by 180ms as shown in Table 7

Table 7 Performance chart of Various Stage of Booting by DMA [16][8]

	firm init	**kernel init**	**user init**	**user splash**	**Total**
Normal Boot	229	660	290	112	1,291
Fast Boot	49	88	287	113	537

The limitation of this technique is the mandatory requirement of specific hardware embedded with DMA controllers such as Fujitsu FR-V Design kit.
One important point to observe here is that this technique can be used in combination with XIP (discussed above) and can provide a counter measure for performance issues in XIP. As we have seen that XIP cause significant performance delays because slow flash memory over RAM. By using DMA, this performance overhead can be avoided as CPU usage will be avoided.

4.1.3 Fast Kernel Decompression

As we have discussed before, in a traditional booting process, the kernel image is stored in flash in compressed form and it must be decompressed before loading and invoking the kernel. In this technique, advanced decompression algorithms are used in order to reduce decompression time during booting. Traditionally, gzip is the technique used my major Linux distributions. Nowadays we have several compression techniques available which provide better compression speed/ratio than gzip [66]. We have performed some experiments using different compression techniques and we will discuss and analyze the experiment results later in the experiment section.
UCL is one of the compression technique focused by major embedded system organizations nowadays. It implements many advanced algorithms which enable UCL to support a number of desirable features. These features include simple and fast decompression, lossless compression and higher compression ratio at the expanse of compression speed but it does not affect decompression speed. [67]
Sony performed a deep analysis of UCL compression and decompression performance in order to speed up boot time on embedded devices. Experiment results show that UCL decompression on sample file system image is 43 % and kernel image is 28 % faster than gzip. [66]
Other popular compression techniques are Lzop and Bzip2. As stated above, we have performed some experiments using these techniques and we will analyze advantages and disadvantages of these techniques based on the experiment results later.

4.2 Kernel invocation speedups

Boot time reduction techniques which are applied to the kernel stage are considered as part of the kernel speedup techniques. If we look back to figure 3, steps involving hardware initialization, starting network and mounting file systems are involved in this stage. The techniques we will discuss in this section will focus on reduction techniques applicable to these steps.

4.2.1 Disable Console

While booting the system with serial console, the speed of printing characters to the console depends on the speed of serial output and drivers. In Serial console, the time of printing output character depended on speed of the serial port but in case of the VGA console, this time depends on processor speed. In case of slow processors, boot time consumption is very high.
Boot time can be improved by disabling console output. As pointed out above, printk output is redirected to serial port or VGA. By disabling the console output, the time consumed by sending output is saved which in turn reduces the overall boot time. [18]

Console output can be disabled by using the "quiet" option in the kernel command line argument. With quiet in kernel command line, with no other arguments, this process will stops the displaying printk output messages during booting [8] .

An experiment result by Mitsubishi and Hitachi [8], described below suggests disabling quiet option saves few hundred milliseconds on boot time. A boot time comparison study is shown below of quiet and without quiet option.
This experiment was conducted on KMC SH Board, using VGA console, 240 MHZ speed and kernel version CELF-1 (040126). The results of quiet and without quiet option were found as 437 milliseconds and 631 milliseconds respectively. By this technique a time saving of 176 milliseconds is made [8].

On the other hand Hitachi performed an experiment on a TI OMAP board, using a serial console, and 168 MHz speed and kernel version CELF-1 (040126). The kernel booted with XIP, CRAMFS root file system, with preset-LPJ. Results with quiet and without quiet option were found as 551 milliseconds and 280 milliseconds respectively. By this technique a time saving of 271 milliseconds is done [8].

By analyzing the two experiments above, we observe that 40-50 (%) reduction in boot time was achieved by using the disabling console technique. This technique is more useful in a case when the system is using a serial console (in above experiments, system with serial console saves 49% of boot time with respect to 40% while using VGA console). Also, one fact we should consideris that the hardware configurations used in above experiments were minimal and hence there is a significant reduction in total boot time. With complex hardware configurations, this reduction may not be that significant. However, a large set of experiments with various hardware configurations are required in order to establish this fact scientifically. This task can be considered as future work.

4.2.2 Avoiding Calibration Delay

One of the initial routine in kernel is calibrate_delay(). This routine is used to determine the most accurate value of a variable loops_per_jiffy (LPJ) by executing a series of delays. LPJ is used to perform short delay cycles during further kernel execution. The performance cost of this calibration depends on the total number of iterations required to calibrate and the length of the iteration instead of the CPU speed. Ideally, value of LPJ remains same for same architectures and same clock frequency [8][14] [72][73][74].

The clock frequency is defined by a variable HZ. Each iteration takes 1 jiffy time, which is define by HZ variable. For 2.4 kernel versions for i386 architecture, HZ value is defined as 100. This makes the length of the jiffy is 10 milliseconds. Normally, total 20 to 25 iterations are required for performing calibration of LPJ; hence total time required is 250 milliseconds. Nowadays, 2.6 versions of kernel are using 1000 as HZ value which in turn makes length of 1 jiffy is 1 millisecond. This reduced total calibration time by 225 milliseconds. But there are still many embedded systems, for which this HZ value is 100, hence bypassing the calibration delay is useful for them. Bypassing of calibration delay can be achieved by hard coding LPJ value. [8] [72]

A series of experiment was performed on LPJ. Samsung performed an experiment by using pre-calculated LPJ value for reduction of loop overhead and saving boot time and saved about 250 milliseconds [8]. Tim bird performed another experiment on x86 desktop machine with Pentium 4 processor having frequency of 3 GHz using Linux kernel version 2.4.20 CELF version. This experiment result showed boot time saving of 268 milliseconds. Jyunji Kondo (of Fujitsu Prime Software Technologies) performed experiment of using preset LPJ on FR-V processor by using kernel version 2.6.6 and by using FR-V FR450 core CPU with 360 MHz frequency and saved 205 milliseconds. Noboru Wakabayashi (of Hitachi) conducted the experiment for preset LPJ on a TI OMAP (ARM-based) system using kernel version 2.4.20 and by using OMAO 1510 CPU with 168 MHz frequency and they showed improvement of about 212 milliseconds [72].
Above experiments shows an average saving of 225 milliseconds. This time is insignificant when talking about big systems having total boot time is hundreds of seconds. On the other hand, this technique is very useful when talking about small pocket size embedded devices such as Mobile phone, GPS trackers etc (These devices has boot time of 2-3 seconds).
The major drawback of this technique we can observe is that hard coded value of LPJ can create problems if not accurate. For example, kernel use this LPJ value as delay while probing hardware devices and it can cause instability of devices during booting if delay is not sufficient for proper device initialization.

4.2.3 Reordering of driver initialization

Device drivers are programs used for interaction among devices and CPU. During boot process, these drivers need to be initialized. For this purpose, each driver has an initialization routine, which is called by kernel during kernel invocation process. Hence, boot time can be saved by optimizing these initialization routines. Device drivers are loaded into the kernel in two ways: static loading and dynamic loading. Static drivers are loaded along with the kernel invocation step and dynamic drivers are loaded once the boot process is complete. This process can be optimized based on device usage. If a device is required during boot process, its driver should be loaded as static driver. On the other hand, if a device is not required at boot process, its driver must be loaded as a dynamic module [14].

Another approach is parallelization of driver initialization which was used by Sony while developing its DSC series of digital camera. Some of the devices in the DSC system are required immediately after boot time such as LCD. Hence their driver initialization routines are called in do_initcalls() function during boot time (these drivers are loaded as static drivers). But some devices need very long initialization time (zoom motor takes 2-3 seconds for example). If this device's driver is initialized during kernel invocation, it can cause significant boot time delay. Hence it is beneficial if these drivers are initialized just at the end of secondary boot loader step (beginning of kernel invocation step). By doing this, zoom motor can be initialized in parallel with other device drivers and hence cause boot time reduction [14][75] . This method is device dependant.

4.3 User-space and application speedups

If we look back to figure 3, all steps after Init process startup come under user space initialization. Hence, the techniques discussed under user space and application speedups will focus on reducing boot time during these steps.

4.3.1 Application XIP

One of the techniques for optimizing application startup speed is Application XIP. This is similar to the kernel XIP technique we have discussed in kernel invocation speedup section. To implement application XIP, the kernel should compile with a file system which support linear storage (blocks are stored contiguously) and uncompressed file storage. One free file system which supports these features is CRAMFS. [8] [79]
In application XIP, when a program is executed, the kernel loads the code segment of the application directly from the flash memory of the file system. In order words, executable files in the flash image are directly mapped to virtual address. Binary code of RAM is directly executed without rebuilding it. In this way, time required to load application code segments into RAM is saved together with the decompression time which in turn reduces the overall boot time [8] [77][78].
An experiment was performed using application XIP on 68MHz ARM 925T TI OMAP 1510 machine and the results are shown in table 8. In case of non-XIP, on second time execution the code segments are fetched directly from the cache (stored in cache in first invocation). Hence, for the second invocation, non-XIP has an edge over XIP. The Application XIP is able to reduce total page cache after boot process by 45 %. [77]
The biggest advantage of XIP is saving of the RAM from copying text segment, which can be a very important aspect for low memory systems.

Also, In addition to the similar disadvantages kernel XIP has, a major pitfall of the application XIP is its dependency on CRAMFS file system.

Table 8 Invocation Results of XIP and Non XIP [76]

Invocation	Non-XIP	XIP
First time	3.195 seconds	1.744 seconds
Second time	2.035 seconds	1.765 seconds

4.3.2 Pre Linking

Linking refers to creating a link between executable image and external shared library routines. Linking can be achieved in two ways: Dynamic linking and static linking. In dynamic linking, name of the shared image is placed on the executable image. Actual linking occurs only during the execution of the image, when both the image and the shared library are loaded into RAM. On the other hand, in case of static linking, all shared library routines are copied into executable image during image creation [81]. But the problem with static linking is that it requires large memory to save the kernel image as all shared libraries are compiled with the kernel image.

Kazuomi's [77] analysis suggests that dynamic linking is a time consuming process. Pre linking is a mechanism which provides a fix up to dynamic linking and reduces significant amount of boot time. [77]

It is a mechanism by which program (kernel image in our case) are linked with shared libraries before their actual execution. Pre linking works by fixing the location of program in virtual memory to referring the correct offsets in the library and allocation of the system program occurred before time [77] [81]. It is a technique to reduce the application initialization time. Since a lot of time is spent in resolving symbols to dynamic libraries. By using Pre Linking, Philips reduced 30 % of application initialization time per application. [81] By using Pre linking, an overall of 71 % application initialization time is reduced. [82]

Pre linking uses an ELF (Executable and Linkable format) image translation to improve initialization time of application by speeding up dynamic linking of the ELF program [82][83]. This mechanism works by assigning a base address to each shared library in the program and symbol resolution is done at link time by base addresses. If a corresponding base address is already used by a shared library other than the assigned pre linked library (which will be the case if a non-pre linked library is loaded into that base address), then all resolution made during link time for that particular library will be terminated at load time. After termination, that library is treated as a normal non ELF library, now base address is determined by usual dynamic linking (i.e. during runtime system resolution occurs). In order to optimize the pre linking process, better coverage of shared libraries should be done. This means, the total number of non pre linked libraries should be minimized.

The experiment results performed with Linux mobile application running Linux version 2.4.x on ARM processor by using dynamic Pre linking show reduction in total time. The time taken without Pre linking during application startup is 2478 millisecond and time using Pre linking is 125 milliseconds [77][81].
The main advantage of Pre linking is fixing all the resolution occurring in the program only once, thus saving the cost of linking at runtime. The disadvantage found of Pre linking is shared library, if they are change, the fixing is not correct, hence fixing of program need to be done again.

4.3.3 **Optimize RC Scripts**

When the Init process starts, it runs different RC scripts in order to invoke various system services. RC scripts are used to perform basic functions of system services. A lot of boot time can be reduces by optimizing these scripts [78]. Now, all scripts in Linux are executed using default shell (which may be bash or ksh). Shells provide different utilities and commands to support scripts intended to run on them. But these shells are very heavy because they consist of a huge numbers of utilities. Hence, there is a performance tradeoff while executing RC scripts using shells. Busy Box is a tool which combines smaller versions of common shell utilities and commands which are classified as built-ins and applets [78][80].

Built-ins are called by class and applets by 'fork/exec' system calls. Apart from this, busy box scripts can also use different external commands and utilities. Busy box Suffers from performance issues due to fork/exec system calls which are heavy and make shell scripts worthless. It is optimized by reducing total number of fork calls and total no of executions by reducing its programs. [80]
RC scripts are optimized by running through busy box instead of a shell. Also, optimization of RC scripts can be achieved by excluding usage of pipes commands, reducing number of commands with a pipe, removal of unnecessary code in the scripts and replacing more external commands by busy box built ins. [80]
Sony research depicts an reduction of about 8 seconds to 5 seconds on Debian RC scripts on OMAP 1510 (ARM 9) processor with 168 MHz [80]. The main disadvantage of this technique is it dependency on scripts.

5. Boot time measuring Tools

Measurement helps us identify key time consuming steps in the booting process. Measurement can be performed by stop watch and have no impact on the system, but it has dependency on a person's reaction time who records the measurement. If reaction time is long, the accuracy of result is a big question mark. Computer aided measurement is always preferred over manual methods due to accuracy of the results. Measurement always requires good measuring methods. Through boot time measurement methodology we will compare different measurement tools among each other to get the best possible methods for measurements.

Boot time measurement analyzes the applicability of different boot time measurement tools. By applying the boot time measuring tool on different booting steps such as kernel and user space initialization, a comparative study was performed and also identification of the best tools to apply at various stages of the booting process was done.

Applicability of different boot time measurement techniques can be classified Into three stages as followed in Table 9 [19] .

Table 9 Area of Measurement and Applicable Techniques

Sr. No	Type of Measurement	Measurement Tool
1	System wide time Measurement	Grab Serial
2	System wide time Measurement	Uptime
3	Kernel invocation time Measurement	KFT
4	Kernel invocation time Measurement	Printk-Times
5	User space initialization time measurement	Strace
6	User space initialization time measurement	Boot Chart

5.1 System wide time Measurement

In the system wide time measurement stage, the boot time of the entire booting process is measured. Following techniques are discussed in this section.

- Grab Serial
- Uptime

5.1.1 Grab serial

Grab serial is a program which captures and shows data from the serial port. The main task of this tool is to collect output messages written on serial console from target machine running Linux and saves records on the host machine [20]. The Grab serial puts a time stamp on each line received from the serial port already specified and displays it on serial console output. By doing this it is useful for reporting timing of low level events of the booting process [21].

By using grab serial command line options, it is possible to specify options which are used on the Linux node, along with port setting. In case there is no option defined, it will use default setting on serial port /dev/ttyS0, at 115200 baud with "8, None and 1" (8N1). The program runs for infinite loops, it reads input from serial port and displays output in the desired way. It provides timing information of each line received. The result generated from grab serial is shown from Tim Bird experiment on Linux Arm machine is shown in appendix 9.5. [20]

By using following script 'show_delta' from the kernel 'scripts' directory the relative time duration between lines can be traced. Data can be saved to a file and can be displayed by following given below.

```
Grabserial -e 20 >/tmp/bootlog.txt; linux_src/scripts/show_delta /tmp/bootlog.txt
```

The main advantages of Grab Serial are shown in the following: [20] [42]

- No need to set up instrument on target machine.
- No effect of slow down on target because of instrumentation.
- It only consumes host CPU cycles.

Disadvantages [20] for this tool are listed below: [42]

- Installation problem, it requires python serial.py because grab serial is a python program and it is not shipped with python.
- Printk messages are piled up in the kernel during early Init process.
- In order to measure the kernel boot up event kernel message has to be turned on.

5.1.2 Uptime

The easiest measurement is achieved by using Uptime. Uptime gives a single line display of information of current time, time duration of system running, shows the total information of users who were logged in and system load averages for last 1,5 and 15 minutes [19] [43]. It gives the time stamp of the full system.

It works by adding the following command to /sbin/init or rc.local.

```
echo -n "uptime:" ; cat /proc/uptime [19].
```

An experiment conducted by Tim Bird (Sony Electronics) [19] with Uptime is shown below in Table 10 on ARM and x 86 configurations.

Table 10 Experiment Result of Uptime

Architecture of Kernel	Configuration	Uptime
x86	nfs fs mount, and short RC script	4.91 3.04
ARM	flash fs mount, short RC script, some services	5.42 1.56

5.2 Kernel invocation time Measurement

The kernel measurement tools determines the time spent in various kernel functions and subroutines during kernel invocation. These tools help us to give an overall analysis of the most time consuming kernel functions and subroutines which are required in order to optimize boot time. Major tools for determining kernel invocation time are discussed below:

- KFT
- Printk -Times

5.2.1 KFT (Kernel Function Time)

KFT is a kernel function tracing tool which provides a timing overview of kernel procedures and subroutines. It helps us perform our analysis in order to find the most time consuming procedures. It uses “-finstrument-functions” capability of gcc complier to add measurement on each callout and exit. KFT can be operated by two mode of operation; dynamic trace configuration and static_run mode configuration. [44] [45]

KFT can be automatically started and stopped the trace, even possible to handle interrupt as part of trace and able to filter the data by various screening parameters. KFT trace data can be generated from /proc/kft-data on completion of trace. In unfiltered mode it can collect a flow of information and help us to debug in order to optimize the code [44] [45].

KFT’s major disadvantage is the overhead generated. KFT harnesses the “finstrument-functions" capability of the gcc compiler for kernel time measurement for function’s entry and exit. The overhead is generated in kernel execution when trace is not active. That is the main reason it is disable in the board as default configuration. In case of time sensitive operations this overhead proved to be costly. The experiment was conducted at Mitsubishi for measurement of overhead of KFI (the predecessor to KFT). The period is from routines start kernel () to smp_init().The result of experiment is shown below in table 11.

Table 11 Experiment Results of Mitsubishi [46]

System Configuration	KFI	Without KFI	Overhead
Platform was: SH7751R 240MHz (Memory Clock 80MHz)	922.419 msec	666.982 msec	27.69%

KFT provides only relative performance analysis of trace data for timing comparison. Kernel performance overhead is a major factor, because compiler adds additional hooks to the entry and exit of each function, which itself is time consuming during execution. It's not good practice to use KFT on production environment. [45]

Local time of kernel functions is calculated by 'kd' for function entry and exit time and then it is subtracted from duration of other functions lifetime. Due to filtering, interrupt handling, or context-switching these numbers may be ambiguous.

KFT traces each and every kernel function. In case no machine operation is going on, kernel functions are executed in several thousands per second. The buffer gets filled and trace stops automatically[46] [47] .

KFT faces a problem of early clock issues. In many platforms the clock used for tracing time duration is not available when kernel begins execution. In this case the function entry and time duration may be incorrect since their execution begins before clock starts [46] [47]. KFT is not able to trace at the low level initialization before kernel startup. [48]

5.2.2 Printk

Printk is a simple program developed to identify the area of Linux kernel to improve booting time. The patch is being added to standard PrintK routine to get additional timing information. This feature has already been integrated with the mainstream Linux version 2.6.11 [51].

Printk helps us see how long Boot operations take place. This method is used to find out long delays in kernel startup while turning on Printk messages. The timing data in seconds begins to display as soon as the Printk messages start. This program shows timing information difference between two routines or shows the time duration relative to specific message. [8] [51]

Printk messages can be started while booting Linux kernel 2.6.11 and adding the option 'time' or Printk='1' on command line. The output generated from Printk messages is shown below in appendix 9.7. By using dmesg option on command line printk data is collected and saved to dmesg >/tmp/bootup_printks. With the use of a utility program, show_delta can read the information from Printk output and display time duration between the Printk messages. [8] [51]

The main advantages of Printk techniques are its simplicity and robustness. It doesn't require any external program or code or any sub routine to obtain boot time measurement. On the other hand there were few drawbacks encountered during the Printk usage. The Printk message option 'dmesg' doesn't display all kernel messages due to fixed size of the kernel buffer. Each time the printk message added some bytes, which eventually exhaust the buffer. The second disadvantage is that the timing resolution is not efficient. Sched_clock() varies from architecture to architecture. In case of ARM architecture timestamps are zero until clock is initialized but in case of x86 timestamps are available immediately. Some platforms have a resolution of 1 jiffy (of 10 seconds duration) which means result of time increment at this resolution only, so it gives imprecise results for Printk-times. The third disadvantage occurred when the Printk options is turned ON. It hangs the machine due to calling of sched_clock too early in the boot sequence. Some platforms don't support early initialization of sched_clock () before any other events such as interrupts or memory initialization. In case of x86 no such problems occurred but with other platform customization of sched_clock is needed [8] [51] [78].

5.3 User space time measurement

User space time measurement depicts the total time spent from starting of init process to the state where system is operational. The different techniques such as Application XIP and optimization of RC scripts can be adopted to reduce boot time of user space, so to examine the effects of boot reduction techniques, user space time measurement tools can effectively measures the boot time [59]. The user space measurement tools discussed here are Boot chart and Strace. [60]

5.3.1 Boot Chart

Boot chart is a tool used for analyzing boot process during user space initialization. It is the tool used for performance analysis and displaying the GNU Linux boot process. Boot chart collects the data about resource utilization and process information during boot process. Later, it is graphically rendered in PNG, SVG encoded chart [57] [58]. The graphical display of the boot chart is shown in appendix 9.8.

Boot chart uses shell scripts into init phase of the kernel. This scripts runs in the background and collect process information, CPU start, disk usage and data from /proc file system. Performance data collected during the boot chart process is stored in the memory and saved in form of log files in hard disk once boot process complete [58].

Disadvantages of boot chart are that it increases the system boot time by a small fraction as an additional script needed to run to collect statistics during the boot process. It works well with the system having enough resources in terms of RAM and greater CPU power but in limited resources system such as a small embedded device, it does not work well. Boot chart has poor performance in terms of scalability. File input/output overhead is high due to large system calls, data collection is the most time consuming step. Lots of processes are required to be executed while logging data to output files [59].

5.3.2 Strace

Strace is a diagnostic, instructional, and debugging tool. It is used to collect the timing information at process level. It determines time spent during application startup. It can calculate system call count and time spent in each system calls. It can trace which operations are causing serious concerns such as multiple opening of a file etc [50] [61] [64]. The sample result obtained from Strace is shown below in appendix 9.9.
The main advantage of Strace is that it can run in parallel with processes, it can trace all results of the system calls made to kernel and even report detailed information in case of memory failure and error while opening a file. Strace helps to identify the area and troubleshoot those parts of the program which cause problems [62]. It is useful for that part of system where source code is not available or recompilation is not possible. [63]

Major pitfalls of Strace measurement is that it does not work for precise measurements. Also, tracing of whole system is not possible [50].

A summary of all measurement tools and programs discussed above are depicted in table 12 below.

Table 12 summary of measurement techniques

Tools	Area of Measurement	Advantage	Disadvantage
Grab Serial	System Wide Measurement	No Slow down effect on target. Consumes host CPU cycles	Installation Problem due to Python Kernel message Piled up in init
Up Time	System Wide Measurement	Provides timestamp of entire system	Measurement starts only once system clocks starts

KFT	Kernel Measurement	Find Boot latencies in early startup	Suffers from Performance overhead Trace Buffer exhaustion
Printk	Kernel Measurement	Robust tool and no overhead.	Measurement requires precise System Clock. Not all Printk messages are displayed
Boot Chart	User Space Measurement	Starts a daemon early in INIT	Overhead of system calls
Strace	User Space Measurement	Count the system call Measure time stamp of application	System Init Problem

6. Experiments

In this section, we will explain and discuss experimentation performed by us. We conducted these experiments on a personal computer with typical boot time of 100-150 Seconds. The hardware configuration of the experiment machine will be discussed in experiment setup section. We mainly focused on techniques which provided us a reasonable ratio of effort required in implementing those techniques and the percentage reduction in boot time. For example, implementing calibration delay avoidance, it requires information about hardware devices delays and it can cause unstable devices as discussed in previous sections. The boot time we save by this technique is around 225 milliseconds which is negligible with respect to the boot time of 100 seconds.
Another major criterion for choosing the appropriate reduction technique is based on the most time consuming parts of boot process. Decompression of kernel and Initrd and execution of RC scripts are the steps which consumes most of the boot time [78]. POST is another step which consumes a considerable amount of the boot time but we can do very little about this as this step is very critical for hardware reliability of the system.
Keeping above facts in mind, we decided to proceed with fast kernel decompression reduction technique.

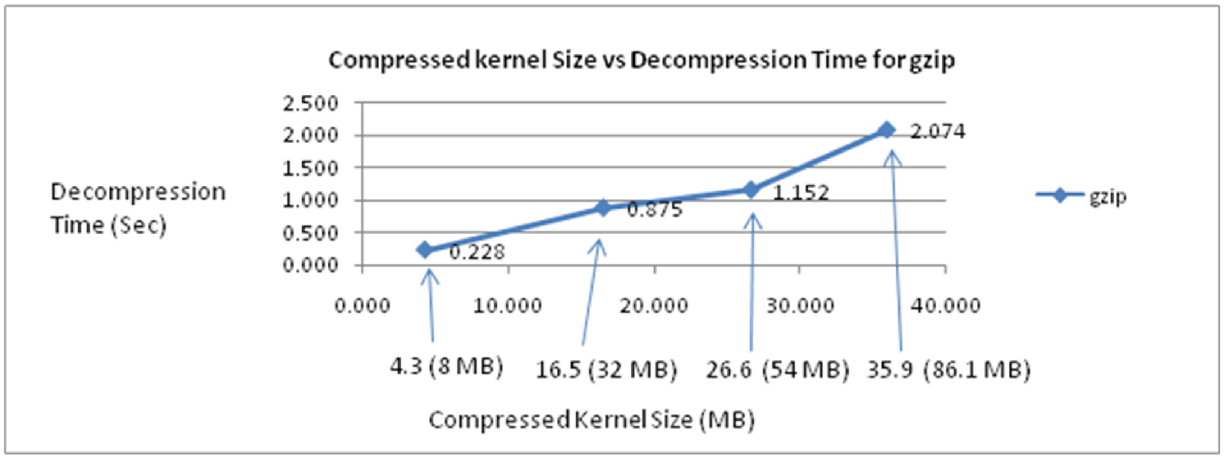
6.1 Experiment setup and goals

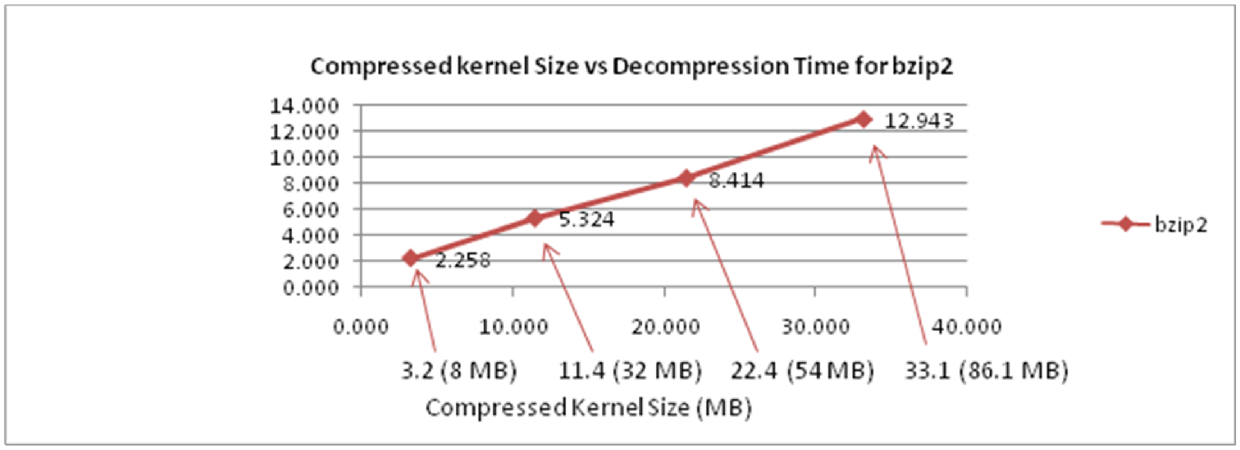
The experiment was conducted on an x86 desktop computer with Pentium 4 processor 3.0 GHz and 3 GB of RAM. The operating system on the machine was Fedora 9 with kernel version 2.6.25.14.

The major goal of this experiment was to find out the performance of various compression techniques and how they affect total boot time of the system. Applicability of these compression techniques for different Linux systems running on different hardware configurations was also analyzed based on the constraints these techniques possess.
Gzip, bzip2 and Lzop are the techniques used during the experiments. Kernel images of different sizes were building with all mentioned compression techniques. Later, these images were used to boot the system and their respective decompression times were noted with the help of Printk measurement technique. The purpose of using different sizes of kernel for decompression is to determine the applicability of compression technique with respect to hardware constraints various systems may have.

6.2 Results

The experiment results are shown in Appendix 9.10 9.11 9.12 9.13 in tabular form. In this section, results are shown with the help of the graph below. This graph will serve as the basis for the discussion in the next section.





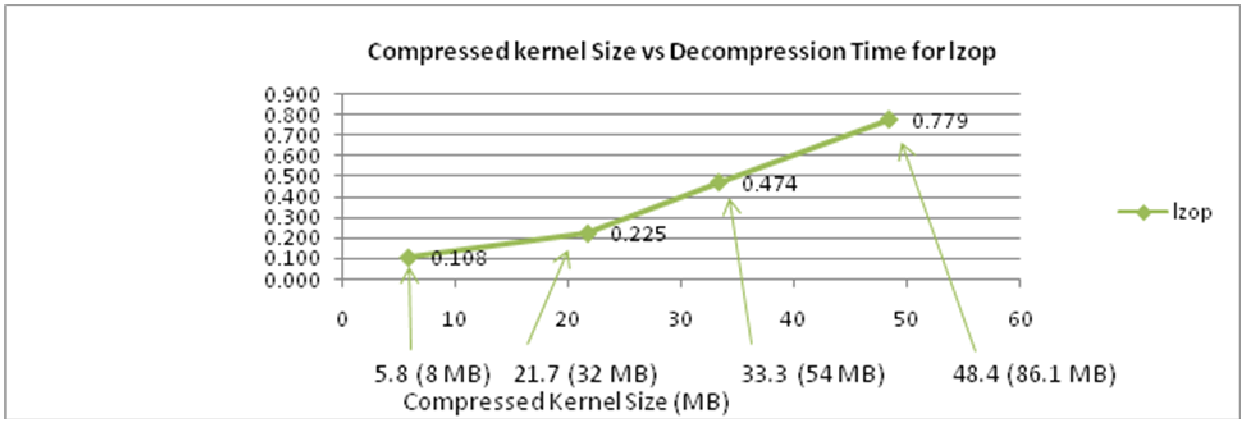


Figure 5 Experiment Result of Fast Kernel Decompression Technqiues

Above graph depicts time taken to decompress various size kernel images. Numbers at the bottom of arrows shows image size after compression and before compression [x(y

MB), x>decompressed image size, y> uncompressed image size]. Dots in the graph show time taken in decompression of images of corresponding sizes.

6.3 Discussion

In this section, we will discuss the performance of each compression technique with respect to the decompression performed over kernel images of different sizes. This will be followed by comparison of all three techniques and their applicability for different system configurations.
In case of gzip as shown from the above graph, Compression ration increases linearly with the rate of 6-8% for initial 3 sizes up to 54 MB. It gets a quick increment of 12% for the fourth image size of 86.1 MB. Overall, gzip gives a good compression/speed ratio slightly towards speed and hence is a good choice for small embedded systems requiring lesser boot time.
Bzip2 does not show very fast decompression speed in comparison to others but provides a very good compression ratio. It also gives a clear indication that for large size kernel images it can further improve the compression ratio which makes it the obvious choice for personal computers having bigger flash memory.
Lzop gives the best decompression speed at the cost of compression ratio but the improvement in boot time by using lzop because of its high decompression speed makes it a clear choice for the systems where minimum downtime is desirable.

Traditionally gzip is being used for kernel decompression in almost all Linux systems. From the above experiments, it is evident that based on system's configuration other alternatives can be looked upon in order to improve the boot time. For real time critical systems where no or minimal downtime is a mandatory requirement, lzop can provide better decompression speed over gzip. Also, if we consider the overall boot process, these are many other files which are saved in flash in compressed form such as file systems. By using an appropriate decompression technique it is possible to reduce the total boot time.

In the above experiment, when we used the minimal kernel of 8 MB with lzop, we got an improvement of around 100 milliseconds over gzip which is negligible. But when we used the default kernel of 86.1 MB, we had an improvement of 1.4 seconds which has a small significance together combining with other reduction techniques. Most of the work these days is being done towards optimizing the user space initialization step in order to reduce boot time. The above experiment shows the possibility to optimize time during the secondary boot loader step with very little effort in comparison to effort required in reduction techniques during user space initialization steps.

7. Conclusion

In the thesis the booting process of the Linux system is analyzed under different conditions. Along with it various areas are discovered to have an ample scope of boot time optimization. Kernel decompression and loading into RAM is one of the areas where significant improvement is possible in terms of time. However, it can cause some extra Flash memory but it can be considered as a reasonable tradeoff for most of the systems. Invocation of background processes and system services are other areas of improvement where traditional method are being used and need to be changed or optimized in order to improve boot time. Replacing busybox with traditional shell utilizes is one of the example.

Chapter 2 goes into details and presents a visualization of the Linux booting process. The entire process is divided into various steps based on the functions performed by each step which allows thinking about each step separately. Along this it is examined what time is spent during each step of the boot process.

After this a question arises what are the different boot time reduction techniques can be considered. Various techniques are categorized based upon their usage in different steps involved in the booting process. Experiments performed by various industry leaders such as Sony, Honda and IBM are analyzed and important results are discussed. It is shown that kernel XIP is one of the techniques which can produce significant results for the systems having big flash storage. Systems which consist of DMA controller can use XIP more effectively if combined with a DMA copy on startup technique.

It is clear from the description of the various reduction techniques in chapter 4 that combining more than 1 technique in different phases of the booting process can increase the advantages and minimize the tradeoffs. This in turn can improve the overall boot time significantly. Reordering of driver initialization can be combined with pre linking and can produce significant boot time improvements.

To calculate and compare different boot times, chapter 5 presents and discusses different measurement methodologies. These methodologies are categorized based on the scope of their usage. The tool Bootchart is useful to gain insight in the boot process and to examine the different factors affecting boot time such as resource utilization. But it does not help to find out the real boot time. Printk is useful if time consumed in each minor step need to be analyzed. It is used in experiments performed to determine the time involved in kernel decompression. Determination of time consumed during these kinds of middle level steps can be a tedious task without help of tools like printk and Grabserial.

Some of the reduction techniques which were initially meant for small embedded systems can also be used in desktops or server systems. Calibration delay avoidance and reordering of driver initialization are the techniques which can decrease the boot time of a desktop or server system only by few seconds but without causing any loss in functionality.

Apart from these, further experiments and research need to be carried out in order to combine and implement various techniques on different systems successfully.

Comparison of various reduction techniques is shown in table below together with their advantages and various tradeoffs:

Techniques	Boot process step	Advantage	Disadvantages
Kernel XIP	Boot loader speedups	Saves Boot Time by Avoiding copying the image from Flash to Ram Avoid Decompression Successfully implemented for Power PC	Performance degrades as Kernel execute in flash. Size increases so suffer Performances
DMACopyOf KernelOnStartup	Boot loader speedups	Can be combined with XIP and solve performance degradation problem.	Hardware dependency as DMA controller is required.
Fast Kernel Decompression	Boot loader speedups	Simple and small modification required	Processor Speed and Memory Constraints
Disable Console	Kernel speedups	Unimportant output messages can cause buffer overflow which in turn affect system's performance.	Error messages can go unnoticed which can affect system's stability
Avoiding Calibration Delay	Kernel speedups	Save unnecessary CPU idle time	Can cause unstable hardware devices
Reordering of driver initialization	Kernel speedups	Devices not required immediately after booting can be initialized later and thus saves boot time	Can cause unstable hardware devices. Also, time improvement is not significant while considering desktops or server systems.
Application XIP	User-space and application speedups	Use Execute-In-Place for applications and libraries.	Requires uncompressed file system. Application performance may be reduced.

Optimize RC Scripts	User-space and application speedups	Uses busybox, smaller shell, built-ins, adjusted scripts	If there is any change or update in shell command or utility, major changes in busybox library are required
Pre linking	User-space and application speedups	Resolves symbols beforehand	Coverage of all shared libraries is desirable. If not done, loses its effectiveness and become similar to normal dynamic linking.

8. Future Work

Apart from reduction techniques discussed in many sections above, there are many other areas where research work is carried out in order to minimize total system downtime without any loss in functionality. Dynamic updates and Kexec are among these areas.

8.1 Dynamic Update

Operating system functionality can be improved by installing kernel features during runtime. Traditionally this is achieved by building a new kernel with new features and functionality and running this new kernel by replacing it with the old one. This causes significant downtime which is not desirable for mission critical real time systems. Dynamic update is the technique with which it is possible to perform software update for resolving bugs or adding new functionality without any down time [89]. It improves not only system availability but also reduces down time, restart and reduce system maintenance time. Dynamic update is implemented by two ways; interposition and hot swapping. An interposition deal with components by fixing the code before or after the original code but the major pitfalls is that it cannot correct the fault in original code. On the other hand, hot swapping works by replacing the original code with new extensions of code and manage the component by direct or indirect swapping [87][88] [89].

In indirect swapping the components is replaced by changing indirect references rather than changing all references. It is simple but runtime overhead occurs due to indirection layer. Direct swapping has no indirection layer between components and its references and it works by replacing old component references with new component references. It is better than indirect swapping since it had no overhead but has more complexity in one way. IBM implemented hot swapping on K42 research operating system by using indirect hot swapping. Hot swapping is dependent on software architecture design such as object oriented and components based environments. In case of Linux it is difficult to implement hot swapping since both Linux kernel and modules are neither object oriented nor component based. Related work is under progress by implementing direct hot swapping on Linux kernel version 2.6.11. The major challenges encountered is the component based architecture since Linux is written in C and component states are easy to identify in C++. A better approach is to work on modules one by one rather than implementing the direct swapping on the entire system. Failure recovery during update is still a major concern. [87][88][89].
From the above discussion it is evident that hot swapping is one of the areas to look for in the future as it can improve system downtime significantly. Its dependency on object oriented and component based architecture is one of the concerns need to work upon in order to have a successful Linux release with hot swapping.

8.2 Kexec

Kexec is another important Linux feature which is important while considering future areas to improving Linux boot time. Kexec is a Linux patch which allows replacing currently the running kernel image with the new updated one without restarting the system. In this method Kexec skips the entire BIOS step of the boot process and directly starts executing the newly loaded kernel image. This results in no involvement of hardware initialization, firmware initialization or boot loader. By using Kexec system restart becomes extremely fast. For mission critical systems, there is a significant improvement in restart related system downtime [90].

The entire process involved in Kexec implementation is extremely complex because it touches many sensitive parts of the system. The biggest challenge during Kexec implementation is to swap the currently running kernel with the new one. This is a known fact that Linux uses a fixed address memory scheme which means that Linux kernel sits in a fix memory location during execution. Hence the major problem is to replace the currently running kernel with the new one while still running in the context of old kernel. Another challenge faced by Kexec is that it can cause unstable hardware devices because hardware test and initialization dost not happen due to the bypass of BIOS step[90].

However, these challenges have been solved for x86 systems and in present time Kexec is the part of the major Linux distributions such as Fedora and Ubuntu [90]. However, a successful running version of Kexec does not exist for other major architectures used in many real time mission critical systems such as Power PC. PPC is the backbone of major telecommunication networks used across the world. Development of Kexec for other major architectures is a big research area which need industry focus.

It can be derived from the above discussion that these are many challenges faced by Kexec. In case of hardware upgrade, Kexec is of no use as it’s not possible to skip the BIOS step in case of hardware changes because it can cause serious system breakdown. Other challenges are graceful termination of processes, devices shutdown and unmounting of file systems [90]. But these challenges indeed provide ample opportunity to focus on this area in order to solve one of the major challenges faced by today’s real time systems which is minimizing the system’s downtime yet fulfilling increasing requirements posed by the customers.

9. Appendices

9.1 POST process of an Award PC-BIOS [54]

Keyword	Description
CPU	Set, verify, Reset, error flags of CPU
Initialize Support Chips	Disconnect video, Parity DMA and NMI, and initialize the PIT, PIC and DMA chips
Refresh	Check capability to refresh PIT chips
Initialize keyboard	Initialize the controller of keyboard
ROM BIOS Test	Generate checksum of BIOS data and compare result with presetting
CMOS Test	Check CMOS chip
Memory Test	Check the first 356ki of memory with routines of the chip set
Cache initialization	Activate external cache
Initialize vector table	Initialize interrupt vectors and install interrupt table in lower memory
CMOS RAM test	Generate checksum of CMOS RAM, if failure then load defaults
Keyboard initialization	Initialize keyboard and NUMLOCK to on
Video test	Check and initialize monochrome and CGA graphic interfaces
Video memory	Check video memory of monochrome and CGA graphic interfaces
DMA test	Check DMA controller and page register
PIC tests	Generate some tests for 8259 PIC chips
EISA mode test	Generate checksum of extended CMOS data, where the information of the EISA interface is placed
Memory size	Write test to all addresses above 256ki with 64ki blocks and init them. If one bit of a block faulty, then this block and all blocks above won't be seen
EISA memory	Initialize all EISA slots and check memory of EISA interfaces
Mouse initialization	Searching for the input device mouse and installing interrupt vector
Cache initialization	Initialize cache controller
Shadow RAM setup	Enable all shadow ram, which are activated by CMOS setup
Floppy test	Check and initialize floppy controller and drive
Hard drive test	Check and initialize hard controller and drive
Serial/parallel	Initialize all serial and parallel ports, use the I/O port information of the CMOS setup to find the ports

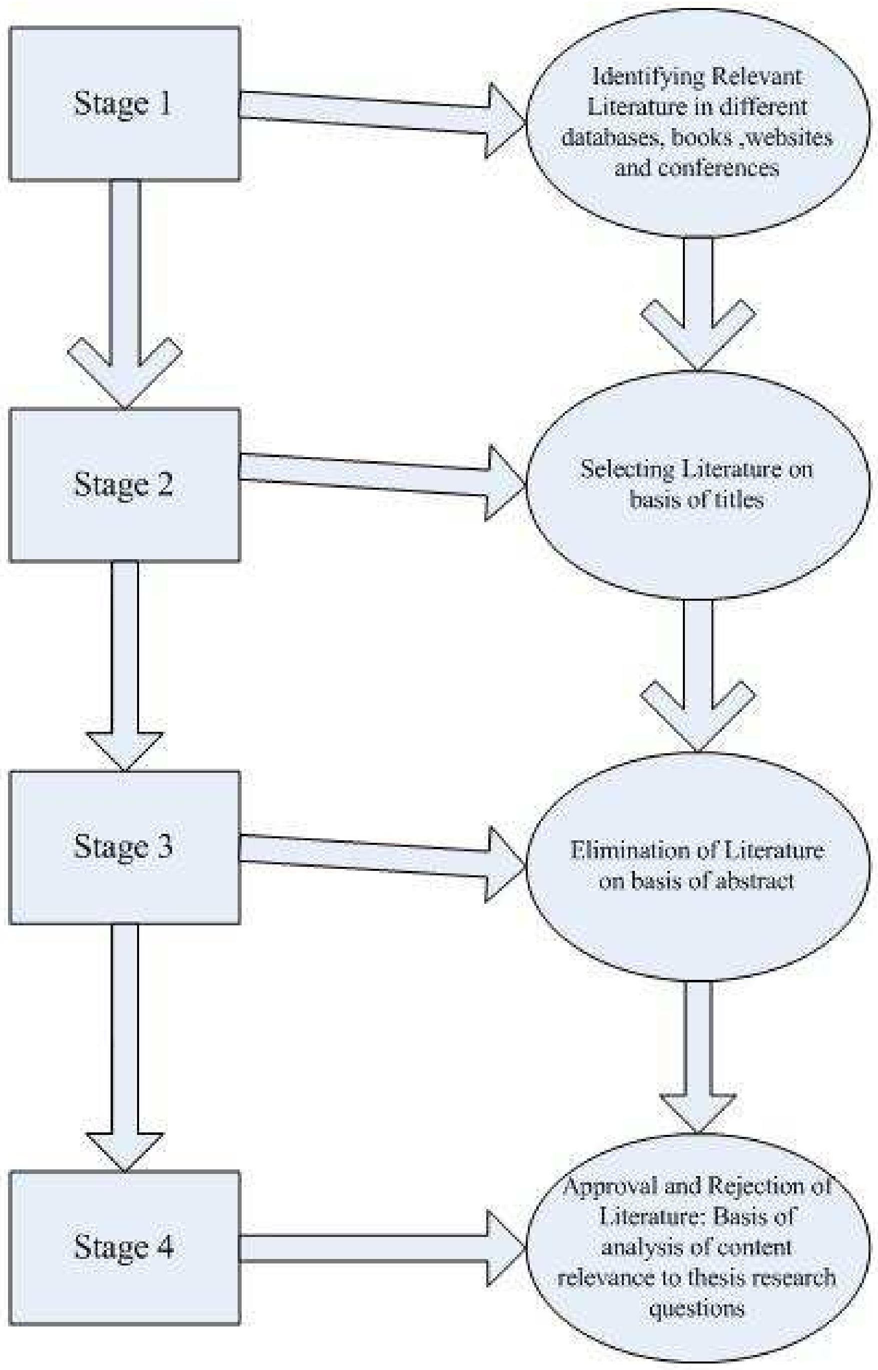
Math coprocessor	Initialize the math coprocessor
Boot speed	Set the default core speed
Manufacturing POST loop	Reboot if loop pin set
Security check	Ask user for password, if one was set
Write CMOS	Write CMOS setup data to ram
Pre boot	Waiting for previous process
Adapter ROM initialization	Initialize all ROMs, which are found between C800:0 and EFFF:F Each ROM checks and initialize all founded devices
Setup time	Set the CMOS time to the entry of the data from address 40 h
Boot system	Give the control of int 19 to boot loader
Boot errors	If the boot loader fails, the BIOS will try to boot from floppy drive

9.2 Literature Search Result

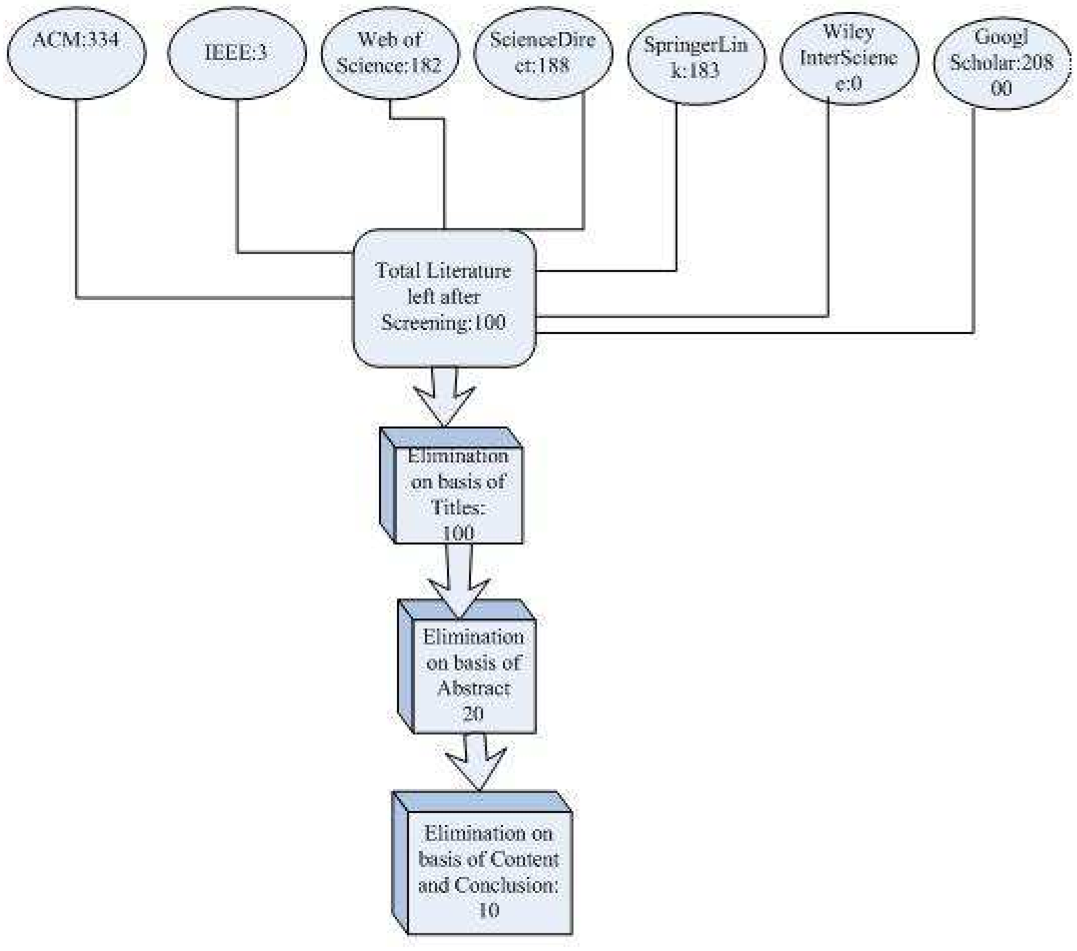
Literature Source	Number of Results on basis of Search criteria			Total Relevant Literature
ACM	Linux : 14847	Linux Boot Time:1387	Linux Boot Time Optimization 334	334
IEEE	Linux: 100	Boot Time Linux:3	Linux Boot Time Optimization: 0	3
ScienceDirect	Linux:13661	Linux Boot Time 838	Linux Boot Time Optimization :188	188
Web of Science	Linux:5100	Linux Boot Time :16	Linux Boot Time Optimization :182	182
SpringerLink	Linux:16085	Linux Boot Time 753	Linux Boot Time Optimization :183	183

Wiley InterScience	Linux:231	Linux Performances:110	Linux kernel Boot: 0	0
Google Scholar	Linux:651000	Linux Boot Time:17800	Linux Boot Time Optimization :20800	20800

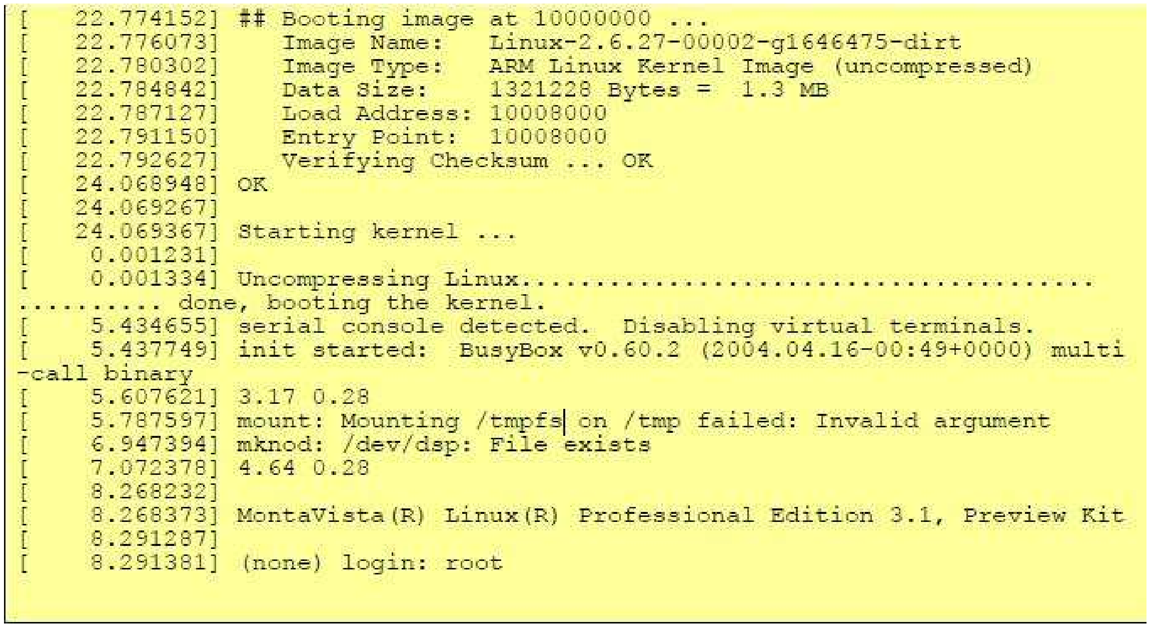
9.3 Search Strategy [39]


Stage 1
Identifying Relevant Literature in different databases, books ,websites and conferences
Stage 2
Selecting Literature on basis of titles
Stage 3
Elimination of Literature on basis of abstract
Stage 4
Approval and Rejection of Literature: Basis of analysis of content relevance to thesis research questions

9.4 Literature filtering Process

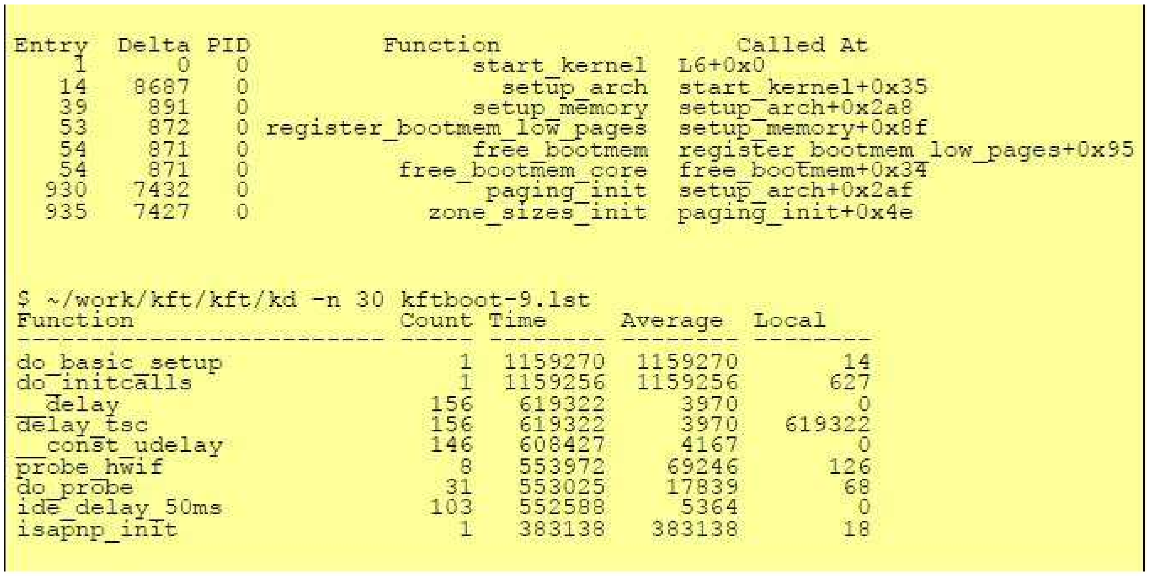

ACM:334
IEEE:3
Web of Science:182
ScienceDirect:188
SpringerLink:183
Wiley InterSciene:0
Googl Scholar:20800
Total Literature left after Screening:100
Elimination on basis of Titles: 100
Elimination on basis of Abstract 20
Elimination on basis of Content and Conclusion: 10

9.5 Sample Result of Grab Serial



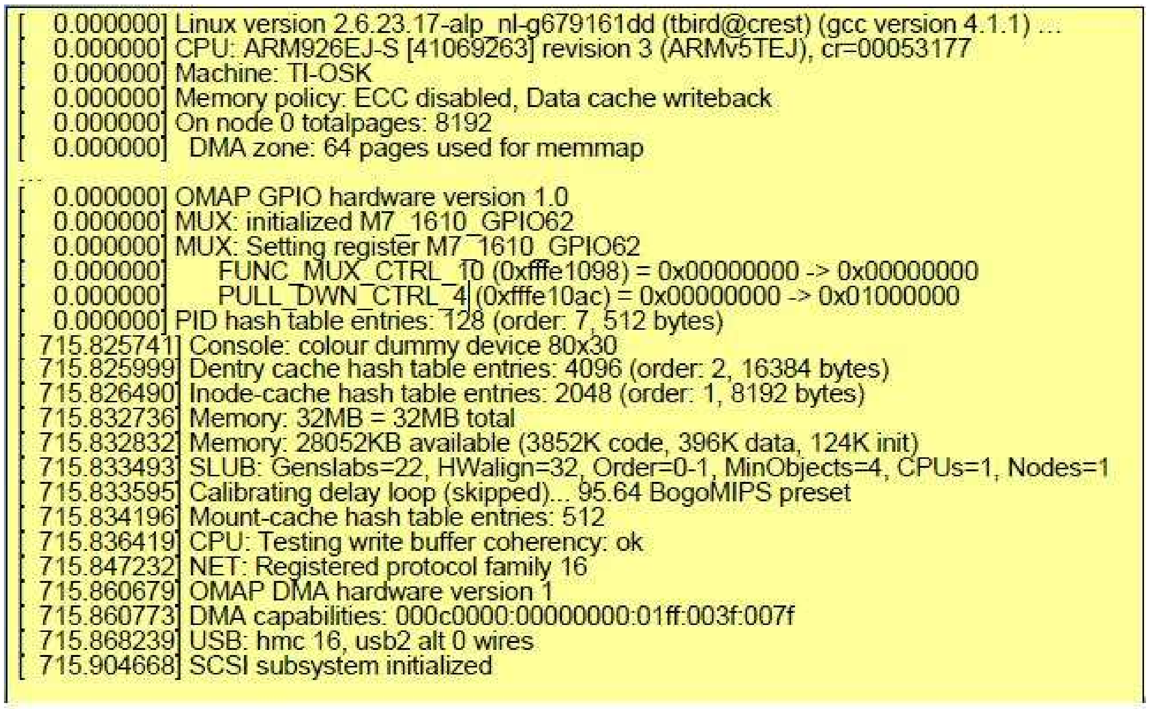
```
[   22.774152] ## Booting image at 10000000 ...
[   22.776073]    Image Name:   Linux-2.6.27-00002-g1646475-dirt
[   22.780302]    Image Type:   ARM Linux Kernel Image (uncompressed)
[   22.784842]    Data Size:    1321228 Bytes =  1.3 MB
[   22.787127]    Load Address: 10008000
[   22.791150]    Entry Point:  10008000
[   22.792627]    Verifying Checksum ... OK
[   24.068948] OK
[   24.069267]
[   24.069367] Starting kernel ...
[    0.001231]
[    0.001334] Uncompressing Linux.....................................
.......... done, booting the kernel.
[    5.434655] serial console detected.  Disabling virtual terminals.
[    5.437749] init started:  BusyBox v0.60.2 (2004.04.16-00:49+0000) multi
-call binary
[    5.607621] 3.17 0.28
[    5.787597] mount: Mounting /tmpfs on /tmp failed: Invalid argument
[    6.947394] mknod: /dev/dsp: File exists
[    7.072378] 4.64 0.28
[    8.268232]
[    8.268373] MontaVista(R) Linux(R) Professional Edition 3.1, Preview Kit
[    8.291287]
[    8.291381] (none) login: root
```

9.6 Sample Result of KFT



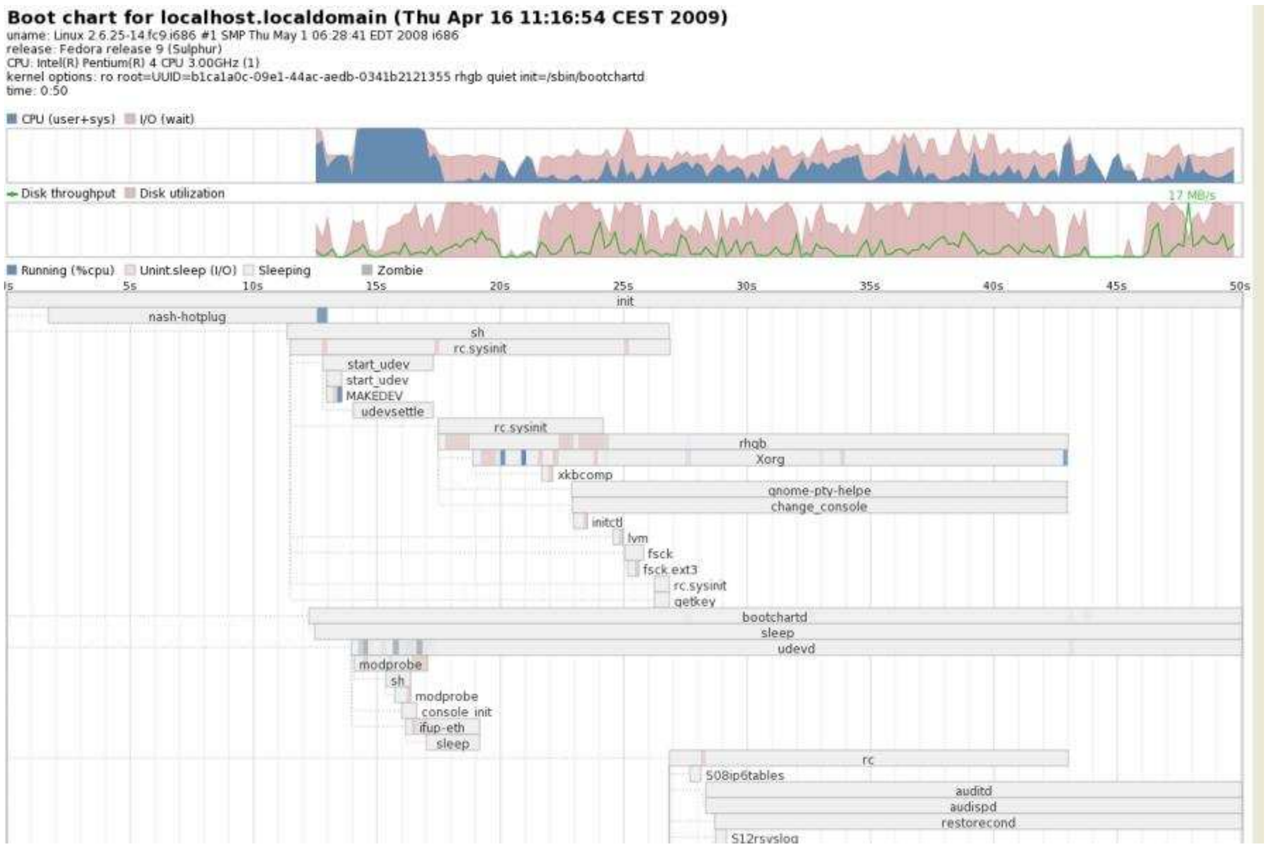
```
Entry  Delta PID          Function                  Called At
    1      0   0               start_kernel  L6+0x0
   14   8687   0                setup_arch   start_kernel+0x35
   39    891   0              setup_memory   setup_arch+0x2a8
   53    872   0 register_bootmem_low_pages  setup_memory+0x8f
   54    871   0               free_bootmem  register_bootmem_low_pages+0x95
   54    871   0          free_bootmem_core  free_bootmem+0x34
  930   7432   0                paging_init  setup_arch+0x2af
  935   7427   0            zone_sizes_init  paging_init+0x4e


$ ~/work/kft/kft/kd -n 30 kftboot-9.lst
Function                   Count Time      Average   Local
-------------------------- ----- --------- --------- ---------
do_basic_setup                 1   1159270   1159270        14
do_initcalls                   1   1159256   1159256       627
__delay                      156    619322      3970         0
delay_tsc                    156    619322      3970    619322
__const_udelay               146    608427      4167         0
probe_hwif                     8    553972     69246       126
do_probe                      31    553025     17839        68
ide_delay_50ms               103    552588      5364         0
isapnp_init                    1    383138    383138        18
```

9.7 Sample Result of Printk

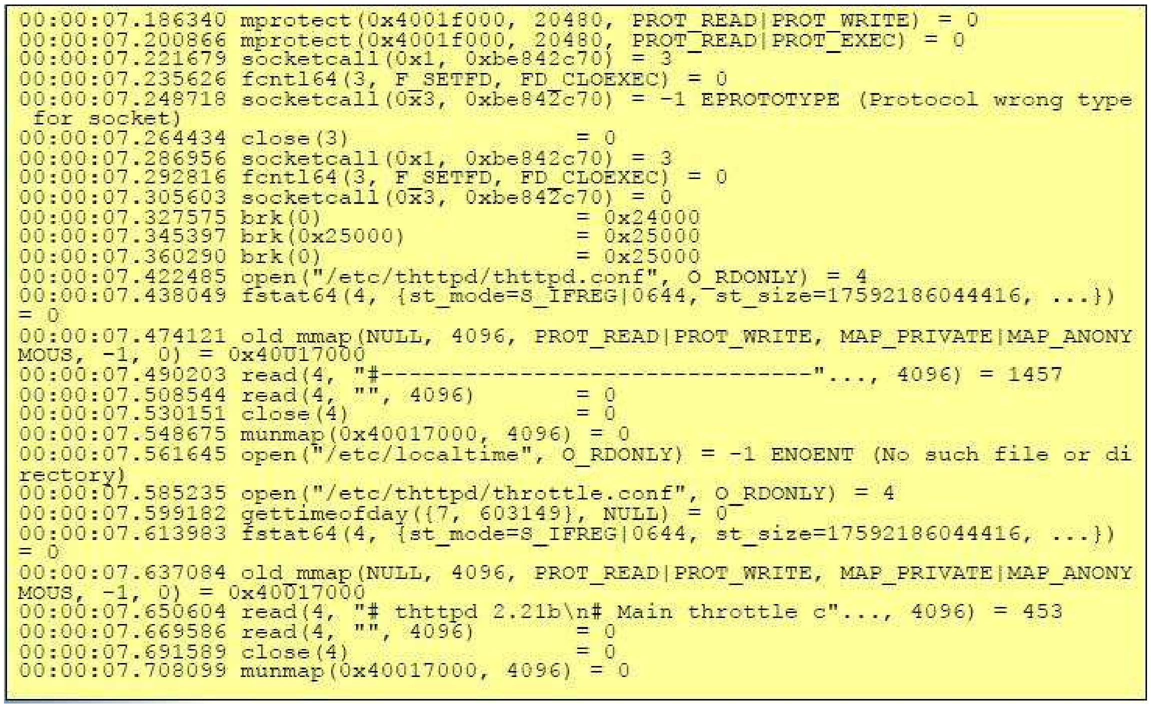


```
[    0.000000] Linux version 2.6.23.17-alp_nl-g679161dd (tbird@crest) (gcc version 4.1.1) ...
[    0.000000] CPU: ARM926EJ-S [41069263] revision 3 (ARMv5TEJ), cr=00053177
[    0.000000] Machine: TI-OSK
[    0.000000] Memory policy: ECC disabled, Data cache writeback
[    0.000000] On node 0 totalpages: 8192
[    0.000000]   DMA zone: 64 pages used for memmap
...
[    0.000000] OMAP GPIO hardware version 1.0
[    0.000000] MUX: initialized M7_1610_GPIO62
[    0.000000] MUX: Setting register M7_1610_GPIO62
[    0.000000]       FUNC_MUX_CTRL_10 (0xfffe1098) = 0x00000000 -> 0x00000000
[    0.000000]       PULL_DWN_CTRL_4 (0xfffe10ac) = 0x00000000 -> 0x01000000
[    0.000000] PID hash table entries: 128 (order: 7, 512 bytes)
[  715.825741] Console: colour dummy device 80x30
[  715.825999] Dentry cache hash table entries: 4096 (order: 2, 16384 bytes)
[  715.826490] Inode-cache hash table entries: 2048 (order: 1, 8192 bytes)
[  715.832736] Memory: 32MB = 32MB total
[  715.832832] Memory: 28052KB available (3852K code, 396K data, 124K init)
[  715.833493] SLUB: Genslabs=22, HWalign=32, Order=0-1, MinObjects=4, CPUs=1, Nodes=1
[  715.833595] Calibrating delay loop (skipped)... 95.64 BogoMIPS preset
[  715.834196] Mount-cache hash table entries: 512
[  715.836419] CPU: Testing write buffer coherency: ok
[  715.847232] NET: Registered protocol family 16
[  715.860679] OMAP DMA hardware version 1
[  715.860773] DMA capabilities: 000c0000:00000000:01ff:003f:007f
[  715.868239] USB: hmc 16, usb2 alt 0 wires
[  715.904668] SCSI subsystem initialized
```

9.8 Sample Result of Boot Chart



9.9 Sample Result of Strace



```
00:00:07.186340 mprotect(0x4001f000, 20480, PROT_READ|PROT_WRITE) = 0
00:00:07.200866 mprotect(0x4001f000, 20480, PROT_READ|PROT_EXEC) = 0
00:00:07.221679 socketcall(0x1, 0xbe842c70) = 3
00:00:07.235626 fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
00:00:07.248718 socketcall(0x3, 0xbe842c70) = -1 EPROTOTYPE (Protocol wrong type
 for socket)
00:00:07.264434 close(3)                = 0
00:00:07.286956 socketcall(0x1, 0xbe842c70) = 3
00:00:07.292816 fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
00:00:07.305603 socketcall(0x3, 0xbe842c70) = 0
00:00:07.327575 brk(0)                  = 0x24000
00:00:07.345397 brk(0x25000)            = 0x25000
00:00:07.360290 brk(0)                  = 0x25000
00:00:07.422485 open("/etc/thttpd/thttpd.conf", O_RDONLY) = 4
00:00:07.438049 fstat64(4, {st_mode=S_IFREG|0644, st_size=17592186044416, ...})
= 0
00:00:07.474121 old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONY
MOUS, -1, 0) = 0x40017000
00:00:07.490203 read(4, "#-------------------------------"..., 4096) = 1457
00:00:07.508544 read(4, "", 4096)       = 0
00:00:07.530151 close(4)                = 0
00:00:07.548675 munmap(0x40017000, 4096) = 0
00:00:07.561645 open("/etc/localtime", O_RDONLY) = -1 ENOENT (No such file or di
rectory)
00:00:07.585235 open("/etc/thttpd/throttle.conf", O_RDONLY) = 4
00:00:07.599182 gettimeofday({7, 603149}, NULL) = 0
00:00:07.613983 fstat64(4, {st_mode=S_IFREG|0644, st_size=17592186044416, ...})
= 0
00:00:07.637084 old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONY
MOUS, -1, 0) = 0x40017000
00:00:07.650604 read(4, "# thttpd 2.21b\n# Main throttle c"..., 4096) = 453
00:00:07.669586 read(4, "", 4096)       = 0
00:00:07.691589 close(4)                = 0
00:00:07.708099 munmap(0x40017000, 4096) = 0
```

9.10 Experiment Result of Compression Techniques of Kernel Size 86.1 MB

Compression Tools	**Kernel Decompression Time (Sec)**	**Original Kernel Size (MB)**	**Reduce Kernel Size (MB)**	**% Decrease in Memory**

gzip	**2.074**	**86.1**	**35.9**	**58**
bzip2	**12.943**	**86.1**	**33.1**	**61.55**
lzop	**0.779**	**86.1**	**48.4**	**43.8**

9.11 Experiment Result of Compression Techniques of Kernel Size 54 MB

Compression Tools	**Kernel Decompression Time (Sec)**	**Original Kernel Size(MB)**	**Reduce Kernel Size (MB)**	**% Decrease in Memory**
gzip	**1.152**	**54**	**26.6**	**50.7**
bzip2	**8.414**	**54**	**21.4**	**60.3**
lzop	**0.474**	**54**	**33.3**	**38.3**

9.12 Experiment Result of Compression Techniques of Kernel Size 32 MB

Compression Tools	**Kernel Decompression Time (Sec)**	**Original Kernel Size(MB)**	**Reduce Kernel Size (MB)**	**% Decrease in Memory**
gzip	**0.875**	**32**	**16.5**	**48.4**

bzip2	**5.324**	**32**	**11.4**	**64.3**
lzop	**0.225**	**32**	**21.7**	**32.1**

9.13 Experiment Result of Compression Techniques of Kernel Size 8 MB

Compression Tools	**Kernel Decompression Time (Sec)**	**Original Kernel Size(MB)**	**Compressed Kernel Size (MB)**	**Decrease in Memory**
gzip	0.228	8	4.3	46.2
bzip2	2.258	8	3.2	60
lzop	0.108	8	5.8	27.5

10.References

[1] M. M. Lehman, J. E Ramil, P. D. Wemick, D. E. Perry, and W. M. Turski ," *Metrics and laws of software evolution – the nineties view* ," In Proc. of the Fourth Intl. Software Metrics Symposium (Metrics'97), Albuquerque, NM, 1997.

[2] Michael W. Godfrey, and Qiang Tu "*Evolution in Open Source Software: A Case Study,*" Software Maintenance, 2000. Proceedings. International Conference on Publication Date: 2000,IEEE

[3] M. M. Lehman, D. E. Perry, and J. E Ramil. Implications of evolution metrics on software maintenance. In Proc. of the 1998 Inil. ConJ on Software Maintenance (ICSM'98), Bethesda, Maryland, Nov 1998.

[4] Linux on the IBM ESA/390 mainframe architecture, 2 Feb ,2009, http://linas.org/linux/i370/i370.html

[5] Who Uses Linux?,Feb 2009, http://www.lugod.org/presentations/ca4h/who_uses.html

[6] Ibrahim F. Haddad ," *Open-Source Web Servers: Performance on a Carrier-Class Linux Platform*,",Feb 2009. http://www.linuxjournal.com/article/4752

[7] Lotte Mygind, Rune Hylsberg Jacobsen and Oskar Swirtun ,"*Introducing Linux and open source* ," , http://www.ericsson.com/ericsson/corpinfo/publications/review/2006_01/files/linux.pdf

[8] Tim R. Bird," *Methods to Improve Boot up Time in Linux* ,"Sony Electronics. Proceedings of the Linux Symposium Volume One July 21st–24th, 2004 Ottawa, Ontario Canada. Available:http://kernel.org/doc/ols/2004/ols2004v1-pages-79-88.pdf.[Jan.30, 2009].

[9] D. P. Bovet, M. Cesati, *Understanding the Linux Kernel*, O'Reilly press, 2002

[10] Boot Time, http://elinux.org/Boot_Time [Accesses Jan 31 2009]

[11] Christopher Hallinan," Reducing Boot Time: Techniques for Fast Booting",MontaVista Software," http://www.mvista.com/download//power/Reducing-boot-time-techniques-for-fast-booting.pdf [Accesses Jan 31 2009]

[12] Christopher Hallinan and Sridharan Subramanian," *Tips and techniques for improving embedded Linux startup time* " MontaVista Software and Freescale Semiconductor" http://www.mvista.com/download/MontaVista-Freescale-fastboot-webinar.pdf [Jan 31 2009]

[13] Red Hat Linux Documentation," *Chapter 3. Installing Red Hat Linux Boot Loader Configuration* ," http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/install-guide/s1-x86-bootloader.html [Accesses Feb 23 2009]

[14] Chanju Park,Kyuhyung Kim,Youngjun Jang and Kyungju Hyun ,"Linux Bootup Time Reduction for Digital Still Camera", Samsung Electronics, " ,Co. Proceedings of the Linux Symposium Volume Two ,July 19th–22nd, 2006 Ottawa, Ontario Canada . Available http://www.linuxsymposium.org/2006/linuxsymposium_procv2.pdf#page=303 [Feb 3 2009]

[15] Kernel XIP ,http://elinux.org/Kernel_XIP Last Accesses Feb 25 2009]

[16] DMA Copy Of Kernel On Startup http://embeddedlinux.movial.fi/index.php?title=DMA_Copy_Of_Kernel_On_Startup&printable=yes [Feb 25 2009]

[17] Jyunji Kondo," *Development of Mobile Linux Open Platform*", Fujitsu Software Technologies Limited April 16, 2008,http://www.celinux.org/elc08_presentations/Development_of_Mobile_Linux_Open_Platform.pdf [March,2009]

[18] Disable Console, http://elinux.org/Disable_Console [6 April, 2009]

[19] Tim Bird, Sony Corporation "*Tools and Techniques for Reducing Bootup Time*" http://elinux.org/upload/d/d2/Tools-and-technique-for-reducing-bootup-time.pdf [Feb,2009]

[20] Grab Serial ,http://elinux.org/Grabserial [March 2009]

[21] Tims Fastboot Tools,http://elinux.org/Tims_Fastboot_Tools [April 2009]

[22] Tatsuo Nakajima,Masatoshi Iwasaki and Shinichi Ochiai, "*Issues for Making Linux Predictable*", Proceedings of the 2002 Symposium on Applications and the Internet (SAINT) Workshops Year of Publication: 2002 ISBN:0-7695-1450-2,IEEE

[23] 'What is Linux?' Linux Online, Inc., http://www.linux.org,2009 [April 2009]

[24] G.Unelb, G. Ambrosinib, H-P.Beck, SCetin'l, T.Conkad, G. Crone, A, Fernandesh, D.Francisb, M.Joos, G.Lehmandb, J.Lopezb, A.MailovJ, L.Mapellib, G.Mornacchib, M.Niculescub, J.Petersenb, L.Trembletb,S .Venezianob, T.Wildishb, Y .Y asd. Using "*Linux PCs in DAQ applications*" , Nuclear Science, IEEE Transactions on Volume 47, Issue 2, Apr 2000 Page(s):109 - 113

[25] James Sanders ," *Linux, Open Source, and Software's Future*", Software, IEEE Sep/Oct 1998Volume: 15, Issue: 5 On page(s): 88-91 ISSN: 0740-7459

[26] Larry Graham ,Christensen O'Connor Johnson & Kindness," *Legal Implications of Operating Systems*", Software, IEEE ,Jan/Feb 1999 ,Volume: 16, Issue: 1 On page(s): 20-22 ISSN: 0740-7459

[27] S.R.Schach, B.Jin, D.R.Wright, G.Z.Heller and A.J.Offutt ,"*Maintainability of the Linux kernel*", Software, IEE Proceedings -Volume 149, Issue 1, Feb 2002 Page(s):18 - 23

[28] Jacob Hallén,Anders Hammarqvist,Fredrik Juhlin,RJ Dataknosult Anders Chrigström, "*Linux in the Workplace*" IEEE Volume 16, Issue 1, Jan/Feb 1999 Page(s):52 – 57

[29] Michael W. Godfrey and Qiang Tu," *Evolution in Open Source Software:A Case Study*" Software Maintenance, 2000. Proceedings. International Conference ,2000 On page(s): 131-142 ISBN: 0-7695-0753-0

[30] Nir Kshetri, Developing with open source software- "*Economics of Linux Adoption in Developing Countries*", IEEE Software archive Volume 21,Issue 1 Pages: 74 - 81 , 2004,ISSN:0740-7459

[31] The Data Compression Modules: zlib, gzip, bz2 ,http://www.linuxtopia.org/online_books/programming_books/python_programming/python_ch33s08.html [April,2009]

[32] gZip, http://www.oreillynet.com/linux/cmd/cmd.csp?path=g/gzip [April,2009]

[33] Osvaldo Cltia and Maria Feldgen ,"*Applying research methodology to undergraduate courses* ", Frontiers in Education, 2003. FIE 2003. 33rd Annual Volume 3, Issue , 5-8 Nov. 2003 Page(s): S3E - 5-8 vol.3

[34] Research Methodology, "Chapter 7 Research Methodology", University of Pretoria http://upetd.up.ac.za/thesis/available/etd-11072001-165433/unrestricted/08chapter7.pdf [March 2009]

[35] Kitchenham, B.A. 2007. Guidelines for performing Systematic Literature reviews in Software Engineering Version 2.3. Technical Report S.o.C.S.a.M. Software Engineering Group, Keele University and Department of Computer Science University of Durham.

[36] Tore Dybå, y Barbara A. Kitchenham and Magne Jørgensen," *Evidence-Based Software Engineering for Practitioners*" Software, IEEE Jan.-Feb. 2005 Volume: 22, Issue: 1 On page(s): 58- 65 ISSN: 0740-7459

[37] Mark Staples and Mahmood Niazi," *Experiences using systematic review guidelines*" Source Journal of Systems and Software Volume 80 , Issue 9 Pages 1425-1437 Year of Publication: 2007ISSN:0164-1212

[38] Mary Shaw, "*What Makes Good Research in Software Engineering*?" In International Journal of Software Tools for Technology Transfer, Vol. 4, No. 1, 2002.

[39] Dyba, T.; Dingsoyr, T.; Hanssen, G.K."*Applying Systematic Reviews to Diverse Study Types: An Experience Report* "Empirical Software Engineering and Measurement, 2007. First International Symposium on Volume, Issue , 20-21 Sept. 2007 Page(s):225 - 234

[40] Haftmann, F., Kossmann, D., and Lo, E. 2007 ," *A framework for efficient regression tests on database applications* ". VLDB Journal. 16(1), 145-64.

[41] Pyserial ,http://pypi.python.org/pypi/pyserial [April,2009]

[42] Grab Serial, http://elinux.org/upload/b/bf/Grabserial [April ,2009]

[43] UpTime,http://linux.die.net/man/1/uptime [April ,2009]

[44] Kernel function instrumentation -tool analysis, http://dslab.lzu.edu.cn:8080/docs/summer_school_2005/kernel_debuging/doc/kfi/analysis_notes
[45] Kernel Debugging tools index Tools - Documentation/Analysis index http://dslab.lzu.edu.cn:8080/docs/summer_school_2005/kernel_debuging/doc/index.html [April 2009]
[46] Kernel Function Trace ,http://elinux.org/Using_Kernel_Function_Trace [March 2009]
[47] KFT ,http://elinux.org/Kernel_Function_Trace [March 2009]
[48] Nicholas Mc Guire ,"*Kernel Function Instrumentation – KFT",* Distributed & Embedded Systems Lab, Lanzhou University December 31, 2006 .http://dslab.lzu.edu.cn:8080/members/hofrat/kft.pdf [March 2009]
[49] Red Hat Linux ," Documentation, Chapter 5. The proc File System" http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/ref-guide/s1-proc-directories.html [March 2009]
[50] Tools and Techniques for Reducing Bootup Time: Tim Bird Sony Corporation
[51] Printk Times ,http://elinux.org/Printk_Times [March,2009]
[52] Parallelize applications for faster Linux booting, http://www.ibm.com/developerworks/linux/library/l-boot-faster/index.html [Jan 2009]
[53] Red Hat Linux Chapter 3. Boot Process, Init, and Shutdown http://www.redhat.com/docs/manuals/linux/RHL-7.2-Manual/ref-guide/s1-boot-init-shutdown-booting.html#S2-BOOT-INIT-SHUTDOWN-SYSV [Jan 2009]
[54] BIOS Kompendium, http://www.bios-info.de/4p92x846/awpost53.htm [Jan 2009]
[55] Wernel Almesberger, "Booting Linux: The History and the Future" ,June 2000, http://www.almesberger.net/cv/papers/ols2k-9.pdf [Jan ,2009]
[56] Datei:Bootvorgang Linux.png http://de.wikipedia.org/w/index.php?title=Datei:Bootvorgang_Linux.png&filetimestamp=20080516145237 [Feb,2009]
[57] Boot Chart ,http://bootchart.org/ [March,2009]
[58] Boot Chart ,http://elinux.org/Bootchart [March 2009]
[59] Visualizing Resource Usage During Initialization of Embedded Systems Matthew Klahn and Moosa Muhammad,Motorolla , http://tree.celinuxforum.org/CelfPubWiki/ELC2006Presentations?action=AttachFile&do=get&target=VisualizingResUsageDuringBoot.pdf [March 2009]
[60] Lightweight boot chart ,Fujitsu Software Technologies Limited Shuuji Miyake, http://tree.celinuxforum.org/CelfPubWiki/JapanTechnicalJamboree16?action=AttachFile&do=get&target=bootchart-lite-en.pdf [March 2009]
[61] Strace, http://linux.die.net/man/1/strace [April 2009]
[62] Strace - A Very Powerful Troubleshooting Tool For All Linux Users :http://linuxhelp.blogspot.com/2006/05/strace-very-powerful-troubleshooting.html [April 2009]
[63] Strace ,http://www.cyberciti.biz/tips/tag/linux-strace [April,2009]
[64] Arnold Robbins UNIX in a Nutshell, O'REILIY Third Edition September 1999 Pages: 616 ISBN 10: 1-56592-427-4
[65] Claes Wohlin,Per Runeson,Martin Höst, Magnus C.Ohlsson,Bjorn Regnell and Anders Wesslén: Experimentation in Software Engineering: An Introduction The Kluwer International Series
[66] Fast Kernel Decompression http://elinux.org/Fast_Kernel_Decompression [March 2009]
[67] UCl ,http://www.oberhumer.com/opensource/ucl/ [April 2009]
[68] lZop, http://www.lzop.org/ [April 2009]
[69] bZip, http://www.bzip.org/ [April 2009]
[70] gZip, http://www.gzip.org/ [April 2009]
[71] Peter Rauschert, Yuri Klimets, J¨org Velten and Anton Kummert, "*Very Fast gzip Compression by means of Content Addressable Memories* " ,TENCON 2004. 2004 IEEE Region 10 ConferenceVolume D, Issue ,Page(s): 391 - 394 Vol. 4

[72] Preset LPJ , http://elinux.org/Preset_LPJ [Feb 2009]
[73] Preset LPJ, http://www.celinuxforum.org/CelfPubWiki/PresetLPJ [Feb 2009]
[74] Calibrate Delay Avoidance Specification
http://elinux.org/Calibrate_Delay_Avoidance_Specification [March 2009]
[75] Reordering of driver initialization
http://elinux.org/Reordering_of_driver_initialization [March 2009]
[76] Application XIP, http://tree.celinuxforum.org/CelfPubWiki/ApplicationXIP. [Feb 2009]
[77] Kazuomi Kato, Tetsuji Yamamoto, Teruto Hirota, Masashige Mizuyama,"*Embedded Linux Technologies to Develop Mobile Phones for the Mainstream Market*", Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE Publication Date: 8-10 Jan. 2006 Volume: 2, On page(s): 1073- 1077 ISBN: 1-4244-0085-6 Current Version Published: 2006-02-13
[78] Linux Kyung Ho Chung, Myung Sil Choi and Kwang Seon Ahn, "*A Study on the Packaging for Fast Boot-up Time in the Embedded Linux*", Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on Volume, Issue, 21-24 Aug. 2007 Page(s):89 – 94 Year of Publication: 2007 ISBN ~ ISSN:1533-2306 , 0-7695-2975-5
[79] Configure Linux for XIP (Execution In Place)
http://www.denx.de/wiki/bin/view/DULG/ConfigureLinuxForXIP [Feb 2009]
[80] Optimize Rc Scripts http://elinux.org/Optimize_RC_Scripts [Feb 2009]
[81] PreLinking http://elinux.org/Pre_Linking [March 2009]
[82] Changhee Jung,Duk-Kyun Woo,Kanghee Kim and Sung-Soo Lim, "*Performance characterization of Prelinking and Preloading for embedded systems"* International Conference On Embedded Software archive Proceedings of the 7th ACM & IEEE international conference on Embedded software , Salzburg, Austria SESSION: Performance table of contents Pages: 213 - 220 Year of Publication: 2007 ISBN:978-1-59593-825-1
[83] J. Jelinek. Prelink. Technical report, Red Hat, Inc., 2004. available at http://people.redhat.com/jakub/prelink.pdf [May 2009]
[84] Shull, Forrest; Singer, Janice; Sjøberg, Dag I.K. ,Chapter 8 Reporting Experiment in software Engineering " *Guide to Advanced Empirical Software Engineering*" (Eds.) 2008, XII, 388 p. 37 illus., Hardcover ISBN: 978-1-84800-043-8
[85] Colin Robson ,"*Real World Research* ",Blackwell Publishing Second Edition- ISBN:, Hardcover ISBN: 978-0-631-21304-8,
[86] Inside the Linux boot process ,http://www.ibm.com/developerworks/linux/library/l-linuxboot/ [Jan,2009]
[87] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski and Jeremy Kerr," *Providing Dynamic Update in an Operating System*" http://ertos.nicta.com.au/publications/papers/Baumann_HADKWK_05.pdf [Feb 2009]
[88] Yueh-Feng Lee and Ruei-Chuan Chang. "*Hotswapping Linux kernel modules*." Journal of Systems and Software Volume 79, Issue 2, Pages 163-175,February 2006
[89] Mark E. Segal and Ophir Frieder. "*On-the-fly program modification: Systems for dynamic updating"*. IEEE Software, 10(2):53–65, March 1993.
[90] http://www.ibm.com/developerworks/linux/library/l-kexec.html