Booting Linux Really Fast

Student Research Project

Author: Daniel Parthey
Author e-mail: daniel.parthey@informatik.tu-chemnitz.de
Date of Submission: 10 April, 2006
Supervisor: Dr. Robert Baumgartl
Abstract

This research project evaluates startup times of the linux boot process from power-on until user login. Time consuming parts of the boot process are investigated and methods how to speed up the whole process are discussed. It includes an analysis and comparison of different BIOSes, the kernel startup sequence and different approaches to start user space services. This project also compares the startup times of everyday desktop systems with the EPIA-ML6000EA Mini-ITX board, an embedded x86-compatible system.
1 Introduction

Modern operating systems are becoming more and more complex and include drivers that support many types of hardware. Almost each additional line of code means an increase of execution time. Even though modern processors are getting faster, the linux boot process does not follow this trend. After power-on of a common linux system there are still dozens of seconds to wait until the system gets ready for use.

The linux operating system has many advantages over other operating systems such as high adaptability and a huge community. Therefore, it would be very attractive to distribute this free operating system with embedded devices. Nevertheless, there are still problems with the acceptance of linux on consumer electronics, because device vendors cannot accept long boot delays. Otherwise they would frighten away their customers. So most vendors are forced to distribute other operating systems with their devices.

This project aims to analyse the linux boot process in order to find out which parts take most of the time and how to reduce the delays. Chapter 2 introduces the topic by giving an overview of the linux boot stages. Chapter 3 presents some existing tools which can be used for analysis of delays inside and outside of the linux kernel and discusses their applicability for this research project. Chapter 4 is divided into three sections where each of them describes an experiment that has been carried out to analyse the boot process. Each section describes setup, realization and finally discusses the results. Chapter 5 discusses some measures which can help to speed up the boot process and chapter 6 highlights some topics which might be the basis for further work.
2 Linux Boot Sequence

To get a better understanding how the linux boot process works, the following sections will shortly describe its stages.

2.1 Machine Reset

When an x86-compatible CPU is restarted or powered on, the timestamp counter processor register (TSC) is reset to zero. This can be used to determine the timespan between any power-on/reset of the machine and a certain measurement point.

2.2 BIOS

The first stage of the boot process on personal computers is usually the Basic Input/Output System called BIOS. The BIOS code is read from a non-volatile memory and is responsible for initializing and checking hardware, directly after the power is switched on or the machine is reset. When the initialization phase is finished, the BIOS loads the boot instructions from the master boot record (MBR), any boot sector on a harddisk or from any other device supported by the BIOS and executes the code which is located there. Furthermore, the BIOS provides basic interfaces to access hardware in the computer.

Some embedded systems do not need a BIOS, because they execute the operating system kernel from a fixed memory address on a non-volatile memory where it must have been written beforehand.

2.3 Bootloader

The bootloader is the second stage of the boot process and at least the first part of it is contained in the boot instructions. It provides the possibility to select the desired operating system. If linux is selected, it reads the kernel from a previously stored address of a data medium or directly from a filesystem if the bootloader includes drivers for this filesystem type. Then the bootloader executes the kernel.

A bootloader is necessary because the whole operating system kernel, which is often several megabytes large, does not fit into the 512 bytes of a boot sector, even though current kernel images are always compressed.

The bootloader optionally loads an initial ramdisk into memory from which a temporary root filesystem can be extracted into volatile memory. This enables the kernel to load any drivers necessary to mount the real root filesystem from a disk attached to a special controller, from a logical volume or even from the network.
2.4 Linux Kernel

The next boot stage is the execution of the operating system kernel. In file `init/main.c` the routine `start_kernel` can be found. This is the first architecture independent part of the kernel where a useful logging mechanism becomes available through the `printk` kernel function. Calls to all main kernel initialization routines can be found there. They include the setup of timers, scheduling and optionally SMP initialization, hardware and software interrupts, console output, memory and buffer management, architecture dependent initializations and some more.

When the execution of these initial routines is finished, the kernel calls the function `rest_init`, which calls the function `init` as a kernel thread. This `init` function cleans up the initialization memory, populates the root filesystem if an initial ramdisk has been provided and looks for an `init` binary to execute. If the `init=` option is supplied on the kernel command line, the given path is used instead of the predefined init binary paths in the kernel sourcecode. If the `init` argument is omitted, the init thread tries to execute the following files in the same order as listed: `/sbin/init`, `/etc/init`, `/bin/init`, `/bin/sh`. If none of them could be executed, the kernel initiates a kernel panic by calling its `panic` function. This function automatically reboots the machine if the `panic=n` option was supplied on the kernel command line with `n` being a whole number greater than zero.

2.5 Init

The init process called by the root thread of the kernel is the root of the process hierarchy and gets a process identificator (PID) of 1. It usually forks itself to create other processes which become children of the init process and get PIDs greater than 1. The init executable can be a script, then its interpreter must be available on the root filesystem, but commonly the init executable is a binary compatible with the used processor architecture.

Under normal circumstances the init process should never exit, it should wait until the runlevel is changed by the system administrator. If the init process returns, the kernel assumes that something must have gone wrong and initiates a kernel panic.

2.5.1 SystemV Init

The SystemV init system has 6 runlevels and a pseudo runlevel S used for initialization.

- S: Single user, Startup and system initialization
- 0: Halt the system
- 1: Single user, rescue system without services
- 2-5: default runlevel, multi user mode with all services
- 6: Reboot the system

Further documentation about runlevels can be found on any Debian GNU/Linux system in file `/usr/share/doc/sysv-rc/README.runlevels.gz`. SystemV init first prepares the service startup in pseudo runlevel S. On Debian GNU/Linux this is done by executing the shell script `/etc/init.d/rc` with S as its first argument. When the system initialization has finished, init switches to the default runlevel. On Debian systems the default runlevel is 2. So init calls the shellscript `/etc/init.d/rc` again, but this time with 2 as its first argument. This causes the script to invoke all executables in directory `/etc/rc2.d` with filenames
starting with an S like in \textit{start}. There are certain dependencies between the started services, e.g. network services must not be started until the network interface is up. These dependencies are expressed by numbers between 00 and 99 appended to the S in the filename. Scripts with a lower number are executed first, followed by higher numbers. A great drawback of this system is that all these startup scripts are executed in a serial way. The consequence is that all their runtimes add up. If a single script’s execution time is longer than usual, this affects the whole boot process. The serialization of the shutdown process works the same, but it uses runlevels 0, 1 or 6 and script filenames which stop services have to start with a K like in \textit{kill}.

\subsection*{2.5.2 InitNG}

The next generation init system [8] developed by Jimmy Wennlund is a full replacement for the SystemV init utility which abolishes the serial execution of services and daemons. It aims to speed up the boot process as well as the shutdown process by starting and stopping services or daemons in parallel. This is achieved by configuring dependencies between the services and optionally even shell script fragments to start and stop the services if the execution of a single binary does not suffice.

On startup, InitNG reads its configuration files from directory \texttt{/etc/initng/} and uses its dependency resolver to determine which services need to be started before some other service. In consequence, each service which is not a dependency of any other service can be started at an arbitrary time during the boot process. InitNG uses the same configured dependencies to automatically determine the order in which services need to be shut down before a reboot or a system halt. It is also capable of monitoring daemons and restarting them if they unexpectedly shut down. Furthermore, it provides a tool \texttt{ngc} to display or change the status of the different services.
3 State Of The Art

This chapter discusses some available instrumentation methods to gather timestamps from the kernel and shows some tools for time measurements in user space when the kernel has already finished booting.

3.1 Kernel Instrumentation

3.1.1 Kernel Function Trace

The Kernel Function Trace [2] is a kernel instrumentation patch which adds hooks to entry and exit points of all kernel internal functions using the gcc feature `-finstrument-functions`. It creates a virtual file `/proc/kfi_trace` where function timing results can be read from and provides tools to process these data and display them in a hierarchical tree view. There is a dynamic configuration which can be inserted through the `/proc` filesystem from userspace on a running linux system, but for kernel profiling during startup, a static configuration has to be used, which must be compiled into the kernel so that it is loaded automatically on startup. The output data contains addresses and no symbols, so the addresses have to be converted back to symbols using the `addr2sym` utility. A Kernel Function Trace provides very high detail, because the timing of every function is measured, but the negative aspect of this instrumentation is a great overhead and intrusiveness, because all functions (and subordinate functions) are slowed down by calling hooks on each function entry and exit point. Mitsubishi measured [3] an overhead of 27.69%.

3.1.2 Linux Trace Toolkit

The Linux Trace Toolkit (LTT) [5] is a kernel event tracer which helps to analyse kernel events using graphical utilities. It measures a small number of discrete events, produces much lower overhead than the Kernel Function Trace and is less intrusive, but the data acquisition is done from userspace using the `tracedaemon` utility. This daemon is run on the target machine and collects the data from a kernel trace module. This means that the toolkit cannot be used to profile kernel startup times, because the trace module needs to be loaded and the user space `tracedaemon` must be able to run, which is not the case until the process scheduler of the kernel has been initialized.

3.1.3 Timepegs

Timepegs [6] is a kernel patch which provides a means of precisely measuring elapsed time between two points within the kernel. For time measurements the Pentium `rdtsc` instruction is used. It creates a graph of visited measurement points with minimum, average and maximum elapsed times between them. The resulting graph can be read from a virtual file `/proc/timepeg` and converted into a human readable form with the help of the `tpt` tool. The caveat of this patch is that timepeg accounting is disabled.
when the system is booted. The first read operation on /proc/timepeg acts as a trigger to turn on timepeg accounting and the timepeg instrumentation itself is initialized during kernel startup, so an accounting can only be triggered from a running kernel, which is rather useless for profiling the kernel boot process.

### 3.1.4 Summary

Each kernel profiling tool has its designated use, but most of them are either too intrusive or inappropriate for profiling the boot process itself, because when the instrumentation code is ready to use, the boot process has already finished. The logical consequence is to develop a simple and unintrusive kernel patch which aims to work very early in the boot sequence and contains as little instrumentation code as possible. The patch `kernel/boot_profile.diff` which will be described in section 4.3.1 uses only the `cpuid` and the Pentium `rdtsc` commands, some variables and the `printk` function, which are all available very early. It gathers some timestamps of certain measurement points inside the kernel startup routine and afterwards writes them into the kernel ringbuffer.

### 3.2 Services Instrumentation

#### 3.2.1 Bootchart

Bootchart [7] is a utility to gather and visualize the time schedule during the start of user space services initiated by `init`. A simple shell script grabs the current process list in regular intervals and as soon as a certain process name like `getty` appears on the process list a small Java or Perl program renders a Gantt diagram out of it. This signalizes that the boot process has finished.

### 3.3 Userspace Utilities

#### 3.3.1 strace

The system call tracer userspace tool `strace` provides an option to display a precise timestamp (microsecond resolution) along with each system call. This allows an estimation of the time between system calls and the total runtime of an executable.

#### 3.3.2 time

The `time` utility provides another way to determine execution times in userspace, but it is not precise enough to do an exact analysis of short timespans, which are very frequent during startup.
4 Boot Analysis

4.1 PC BIOS and GRUB bootloader

4.1.1 Methodology and Setup

To analyse the influence of BIOS settings on boot delay, the timespan between reboot and the kernel start has been determined. Since the timestamp counter processor register is reset to zero on reboot, it suffices to read the timestamp counter once when the kernel starts. The value read from the timestamp counter contains the number of CPU cycles between reboot and kernel start.

Reading and printing the timestamp counter has been accomplished with the following kernel patch, which can be found in `bios/patches/tsc_at_kernel_start.diff` and must be applied to the kernel first:

```
--- linux-source.orig/init/main.c
+++ linux-source/init/main.c
@@ -429,6 +429,11 @@
    * Interrupts are still disabled. Do necessary setups, then
    * enable them
    */
+ /* Print TimeStamp Counter CPU register */
+ unsigned long long now;
+ rdtscll(now);
+ printk(KERN_INFO "%llu start_kernel()\n", now);
+ lock_kernel();
    page_address_init();
    printk(KERN_NOTICE);
```

This patch declares a local variable `now` in function `start_kernel` of `init/main.c`, reads the timestamp counter register using the architecture dependent macro `rdtscll` and writes it into the kernel ring buffer using the `printk` function. The timestamp can be retrieved from the kernel ring buffer by calling the `dmesg` command later.

When the kernel is installed, the bootloader must be configured correctly. A template configuration for the grand unified bootloader (GRUB) can be found in `boot/grub/menu.lst`. The configuration should look similar to the following one and must be adapted to the target platform. The configuration options will be explained below.

```
default 0
```
timeout 0
title Linux Test
root (hd0,4)
kernal /vmlinuz ro root=/dev/ROOT_PARTITION init=/init panic=1

The default option must be set to zero to select the first kernel entry. To minimize the bootloader delay, the timeout for the operating system selection must be zero to make the bootloader use the default entry without waiting for any user interaction. The root option defines the root partition where GRUB searches the filesystem for its own files as well as the given kernel and optional initrd images. Harddisk drives and partitions are counted from zero, this means that hd0 corresponds to the first harddisk and 4 corresponds to the fifth partition. The kernel option must point to the image of the patched kernel, and the following options must be passed at the command line: the root argument points to the device which will be mounted as the root filesystem, the init option points to a modified init script and argument panic=1 ensures that the kernel automatically reboots the system after each measurement.

The shell script ./bios/pcbios/init must be copied to /init so that the kernel is able to execute it. This script checks the root filesystem, mounts it in read-write mode and counts the number of measurements already done. If there are pending measurements, it extracts the start_kernel() line from the output of the dmesg command and appends it to file /tsc. If the desired number of timestamps has been collected, it stops and waits for the user to press return. After each run, it also provides the option to obtain an interactive rescue shell by pressing return during a three-second grace time. If the user does not interact, the script mounts the root filesystem read-only again, to force the datafile to be written to disk. Afterwards, it exits to initiate a deliberate kernel panic. This way, all measurements for each single BIOS configuration can be done unattended, no startup scripts or services need to be loaded and the computer automatically reboots after each measurement run. The only user interaction necessary is to reconfigure the BIOS settings each time when another configuration should be investigated. This indeed takes a lot of time and has been done manually, because there is no known method to change specific CMOS settings of proprietary factory BIOSes from a running Linux system.

### 4.1.2 Measurements

After setting up this test environment on an AMD Athlon 64 Winchester 3000+ on an Asus A8V Deluxe mainboard, the AMI BIOS was configured to some kind of worst case configuration by setting all options and features so that they would consume as much time as possible. For example, all hardware was enabled and the quickboot feature was disabled. Some BIOS options have to be set to a fixed value, e.g. the Cool’n’Quiet feature must be disabled to make sure that the processor frequency is not changed dynamically, depending on system load. Otherwise, the same number of processor cycles would not always correspond to the same timespan and the conversion of cycle counts into seconds would yield misleading results.

After the BIOS had been set up, the machine was rebooted several times and the timespans from reset to kernel start were recorded to disk. The result of the first run was discarded, because the first value often was lower than all others and would have skewed the arithmetic mean. Thus, the machine had to be rebooted eleven times to get ten values under comparable conditions.
The next step was to optimize the BIOS configuration and the boot delay by incrementally changing a single BIOS option to a better value, e.g. the MIDI port was disabled, then the parallel LPT port was disabled, etc. For each new BIOS configuration where only one BIOS option was changed, the computer was rebooted eleven times and ten timespans between CPU reset and kernel start were recorded again. The same procedure was done for most of the BIOS options which were thought to have an influence on boot speed.

The average cycle counts from reset to kernel start have been visualized in figure B.1. Its legend contains the BIOS options that have been changed in each step.

4.1.3 Discussion

The first line of table A.1 and the leftmost bar of diagram B.1 show that the worst case BIOS configuration (including the startup time of the GRUB bootloader) needed about 47s until the kernel started. The rightmost bar shows that by only changing the BIOS configuration, this delay could be reduced to an approximate value of 19s which is less than half of the time in the worst case BIOS configuration.

This means that the PC BIOS configuration has a great impact on bootup speed. It is always a good idea to care about BIOS options and do measurements to find out which BIOS firmware and which configuration is the best one for the desired target system. Alternatively one could try to use hardware without a BIOS, but this will not be an option for most desktop computers. Another option is to use LinuxBIOS [9], which will be discussed later.

To find out which BIOS options are the most important ones, one should take a closer look at the single bars of figure B.1 and the arithmetic mean values in table A.1. Disabling external ports like MIDI/game, parallel and serial ports or firewire did not have any significant effect. Switching off the network boot and the networking feature itself had almost no effect. But disabling the redundant array of independent disks (RAID) controller saved about ten seconds instead, hence anybody who does not need RAID should disable this option. Switching audio off, disabling PCI card setup ROM INT19 hooks and the "Press DEL to enter setup" message did not significantly change the startup time. An important setting was to enable the "Quick Boot" option, which saved about eleven seconds by omitting some power-on-self-tests and a ten second countdown intended for reading the mainboard info box which displays cache size, detected harddisks, etc. Disabling the floppy boot saved another second, because the system did not check the floppy drive for a bootable floppy disk anymore. Disabling booting from CD-ROM saved another 100ms, most probably for a similar reason, but the CD-ROM was much faster in detecting that there was no CD in the drive. Surprisingly, after the floppy disk drive had been completely disabled, the boot time drastically increased by about 6 seconds again. Maybe some piece of code waited for a floppy drive and timed out if it could not find one. So the floppy drive was enabled again, to get the better startup time of 20.6 seconds which had been achieved before. Disabling the CD-ROM neither saved nor wasted any significant amount of startup time. After the Universal Serial Bus (USB) port had been disabled, the startup time decreased by one and a half second, so it might be helpful to disable USB in cases where no communication over USB is necessary. The last two BIOS options which were evaluated, MDA resources and APCI/APIC, only saved about 100ms when disabled and so did not have great influence on startup time.

To summarise, a good AMI BIOS configuration for fast system startup times on a standard PC sys-
tem uses the following settings: disable RAID, enable "Quick Boot" or disable "Power On Self Tests" (POST), enable floppy drive, disable USB and any other components that are not needed to run the desired applications on the target system.

4.2 Linux BIOS and FILO bootloader

The long BIOS startup delay of the PC AMI BIOS and the strange experiences with the floppy drive were a motivation to look for ways to get rid of a proprietary BIOS which spends too much time on probing and initializing hardware or waiting for timeouts. LinuxBIOS [9] is a portable (because it was mostly written in C) and open source BIOS replacement which takes advantage of the fact that current Linux kernels already provide functions to initialize and access most of the hardware of today’s desktop computers and some embedded systems. So LinuxBIOS can save time by only initializing core components like main memory, processor and some hardware necessary for loading the linux kernel. Searching the PCI bus for devices before loading the kernel is inevitable in most cases, because the device where the kernel is loaded from (IDE, Network Adapter, Flash ROM, etc.) is often connected to the PCI bus.

In LinuxBIOS terms a small 'payload' can be included with the LinuxBIOS image. This works as long as the base image and payload fit together in the BIOS flash ROM. Unfortunately, a complete linux kernel would not fit in a BIOS flash ROM of about 256 or 512kB. The payload can be any statically linked executable in the ELF format, but the most common payload is FILO [10], a tiny bootloader which loads the linux kernel from an attached storage device.

4.2.1 Methodology and Setup

LinuxBIOS was compiled for and installed on a VIA EPIA-ML6000EA Mini-ITX board following the steps described in the EPIA-M HOWTO which is located in directory HOWTO/ of the LinuxBIOS sourcecodes.

It is important to note that there are some differences between the M and the ML model of the VIA EPIA board. A patch was created to get the EPIA-M target of LinuxBIOSv2 revision 2169 working on the EPIA ML model. It must be applied as follows:

```
patch -p1 < ./bios/linuxbios/install/patches/epia-ml.diff
```

First, this patch suppresses any output on console and serial line for speed reasons. Second, it disables firewire, cardbus and second riser slot support, because none of them are available on the ML model. Finally, the PCI device ID of the VGA chipset detection was changed from 0x3123 to 0x3122 in order to make the write protection of the VGA BIOS work correctly. Before compiling LinuxBIOS, be sure to set the locale to C or else the `make` command will fail on machines with a different locale than English:

```
export LC_ALL=C
make
```

Additionally, it was important to prepend a 64 kB large factory VGA BIOS image from VIA technologies to the LinuxBIOS ROM image in order to get the graphic display working. Usually, the VGA BIOS image can be extracted from memory address 0xC0000 of `/dev/mem` in a running linux system which
was booted from an original VIA BIOS image, but the only VGA BIOS image known to work with LinuxBIOS on EPIA ML is contained in the EPIA-M factory BIOS v1.13 [11], and this image cannot be booted on EPIA-ML models. So the VGA BIOS image was directly extracted from the file using the Award BIOS Editor [12].

The extracted EPIA-M VGA ROM is named `MVPSD_15.rom`, is 64000 bytes large and starts with the two magic bytes 0x55 0xAA. Because the size of the VGA ROM image is less than the necessary 1024*64 = 65536 bytes, it must be padded with 1536 zeroes at the end so that its filesize is exactly 64kB and LinuxBIOS correctly ends up at memory position 0xD0000 when booting the system:

```
dd if=/dev/zero bs=1536 count=1 >> /video.bios.bin
```

FILO [10] was used as the bootloader, to load the Linux kernel from an IDE harddisk. The PCI brute-force scan and unused filesystem drivers (except IDE and ext2) were disabled in the FILO configuration. FILO’s AUTOBOOT_DELAY option was set to zero in order to continue loading the kernel image directly after FILO has started. The following kernel command line was used for FILO’s AUTOBOOT_FILE configuration option:

```
hda1:vmlinuz-filo root=/dev/hda1 panic=3 lpj=2671530 quiet
```

A minimal Linux configuration 
`/bios/linuxbios/install/kernel/config-2.6.14.2-via-epia` was prepared for the EPIA-ML board and a static kernel was built. A working Linux root filesystem was installed on `/dev/hda1` and a symbolic link `/vmlinuz-filo` was pointed to the kernel image in order to match the path given in FILO’s AUTOBOOT_FILE option.

The loops-per-jiffy (lpj) number in the kernel command line must be replaced by the value which is output by the kernel when booting it without the `lpj=n` argument. This presets the loops_per_jiffy parameter and prevents the execution of the delay loop calibration on every boot. The delay loop calibration takes at least the time of initializing one programmable interrupt timer (PIT) channel, because this is used as a time basis.

The VGA BIOS and the LinuxBIOS image which includes the bootloader FILO were written into the BIOS Flash-ROM using the hot-flash method. This means that Linux was booted under the original factory BIOS, then the flash chip was carefully pulled out using an appropriate claw and replaced by an empty flash chip of the same specifications. Then the `flashrom` utility from the `util` directory of the LinuxBIOS sourcecodes was used to write the previously prepared LinuxBIOS image to flash and the system was rebooted.

After the factory BIOS had been successfully replaced with LinuxBIOS, the kernel was instrumented with the developed patch `/kernel/boot_profile.diff` and then rebuilt. This patch writes timestamps into the kernel ring buffer during kernel startup. A script `/bios/linuxbios/install/instrumentation/measurement` which saves the timestamps from the kernel ring buffer to disk was installed into file `/etc/rc2.d/S99measurement` in order to be automatically run on each startup.

### 4.2.2 Measurements

The described setup was used to analyse BIOS startup times on our embedded VIA Mini-ITX board in four different scenarios.
4.2.2.1 Factory BIOS and GRUB bootloader

The first scenario was run with the factory BIOS and bootloader GRUB. The machine was rebooted 100 times and the gathered timestamps were used to determine the minimum, maximum and average delay from power-on until start of the kernel main routine. The harddisk always kept running and power was never switched off.

4.2.2.2 LinuxBIOS and full-featured FILO bootloader

The second and following scenarios use LinuxBIOS and the FILO bootloader. This scenario uses FILO in its full featured standard version, which means that PCI, serial debug console and all supported filesystem drivers were enabled in FILO’s configuration. The linux kernel was not switched to quiet mode, but this does not matter, because the measurement timestamp is read on kernel start. Again, the machine was rebooted 100 times, the harddisk was kept running and the same statistic values as in the first scenario were gathered.

4.2.2.3 LinuxBIOS and optimized FILO bootloader with harddisk already up

In scenario three, LinuxBIOS and the FILO bootloader were recompiled in a minimum configuration where console, serial console and all unnecessary device drivers and hardware detection mechanisms were disabled. A patch bios/linuxbios/install/patches/epia-ml-quiet.diff has been created for LinuxBIOS r2169 in order to completely disable any serial console initialization or output. This patch was applied and LinuxBIOS was rebuilt. Unfortunately, the attached VGA BIOS image or LinuxBIOS were not able to correctly reinitialize after a soft reboot of the linux kernel, so for each run the machine had to be either manually reset using the reset button or completely powered off to put it into a defined state. The machine was rebooted ten times without powering off and the harddisk was kept running.

4.2.2.4 LinuxBIOS and FILO bootloader with harddisk spin-up

The first three scenarios all used a spinning harddisk. The fourth scenario was intended to combine all optimizations of the second and third scenario with a real-world situation where machine and harddisk are first powered off and then switched on for boot. After power-on, the harddisk had to spin up, in the meantime LinuxBIOS and FILO came up and loaded the linux kernel as soon the harddisk got ready to serve requests. As in all the other scenarios, a timestamp was recorded at kernel start. After linux had come up, the machine was halted and powered off again. This procedure was repeated ten times to find out the delay from power-on until kernel start.

The timestamps from the four scenarios were collected in single files for each run. Afterwards, the minimum, arithmetic mean and maximum values were calculated for each scenario using the script bios/linuxbios/data/csv.py. Finally, these timestamp values were converted from cycles into seconds with the help of the processor frequency value from /proc/cpuinfo and the resulting timespans were visualized in figure B.2 using OpenOffice.org.
4.2.3 Discussion

The first scenario on the VIA board as described in section 4.2.2.1 with its 12 second delay was already faster than an optimized AMI BIOS configuration with GRUB on a desktop PC (compare figure B.1 and B.2). Despite of this experience, even shorter startup times could be achieved in the next scenarios.

Table A.2 and figure B.2 show that the second scenario saved almost 5 seconds by only exchanging the Award BIOS by LinuxBIOS and the bootloader GRUB by FILO. The delay between reset and kernel start was only 7.8s from now on, which means a reduction of 39% in comparison to the first scenario.

A slim LinuxBIOS and FILO in the third scenario shortened the delay by another reproducible 200ms (± 10ms) to 7.6s, which means a reduction of only 1.5% when compared to the first scenario. This was not a really great improvement, but eliminating unnecessary sourcecode and drivers from BIOS and bootloader obviously contributes to speeding up the boot process.

Scenario 4 showed that spinning up the IDE harddisk delayed the boot process by 3.77s. Nevertheless, spinning up the harddisk and loading the linux kernel using an optimized LinuxBIOS together with an optimized FILO bootloader needed only 11.37s (as shown in the last line of table A.2) which is even faster than scenario one where the kernel was loaded using the factory BIOS and bootloader GRUB after a simple hard reset with an already running IDE harddisk.

LinuxBIOS can load the FILO bootloader shortly after power-on, even before the harddisk is ready, because the static FILO executable is written into flash as the LinuxBIOS payload. The only remaining problem is the linux kernel itself. A usual linux kernel does not fit into a 256 kilobyte flash, not even if very well configured. So there is the need for an extra device where kernel and root filesystem can be read from very quickly after power-on. This means that flash memory should be preferred to harddisks when loading the operating system on devices where boot time is critical.

Furthermore, LinuxBIOS has a much lower overhead than most factory BIOSes. First, it only contains the necessary code to initialize the system and graphics. Second, it is open source and therefore modifiable. If there is any unnecessary code, it can simply be removed by someone with C programming skills.

The conclusion is that one should use a small LinuxBIOS that is optimized for the target platform and a fast device to load the kernel and the root filesystem from. Any data needed for the boot process should be read from this device and written to RAM if necessary. Application data can be read from and written to a mounted harddrive later, for example for digital video recorders, but using a harddisk to quickly boot a system is definitely not recommended.

4.3 Kernel

BIOS and bootloader are directly followed by the kernel during boot, so the next step was to investigate how kernel internal delays influence system startup time and how much time is consumed by the kernel initialization routines.

4.3.1 Methodology and Setup

The kernel startup sequence was profiled on a common desktop PC with an AMD Athlon64 3000+ CPU running at 1800 MHz and an AMI factory BIOS. The BIOS was configured to support all hardware of
the test system, but any unnecessary hardware or features which were not supported by the test system have been disabled in the BIOS setup. The CPU frequency adjustment "AMD cool & quiet" was kept disabled in the BIOS settings to keep the CPU at constant speed and to be sure to get correct values when converting cycle counts to timespans.

Measurement intervals during kernel startup are much shorter than BIOS startup times. In order to provide the necessary accuracy of a few milliseconds, a cpuid barrier was added before the rdtsc assembler operation. This flushes the processor pipeline and prevents the CPU from reading or executing several opcodes out-of-order which might result in an out-of-order execution of the rdtsc measurement point and commands whose execution times should be measured. The architecture dependent kernel macros rdtsc11 were duplicated and the cpuid barrier was then added to the newly created macros rdtsc11_with_barrier. All necessary changes can be done to the kernel by applying the patch ./kernel/boot_profile.diff which has been created for linux kernel version 2.6.14.2 from kernel.org. The initial ramdisk support was disabled in the kernel configuration and any code necessary to mount the root filesystem was compiled into the kernel. The kernel configuration file ./kernel/kernel-config-amd64-default used here is based on the Debian GNU/Linux standard kernel configuration which enables a lot of features, but compiles almost everything as a module.

During kernel startup, several timestamp counter values are collected in static variables and then printed to the kernel ring buffer. The kernel init= argument must point to the init replacement shell script kernel/init. This script mounts the root filesystem in read-write mode, extracts the timestamp lines starting with 'TS_' from the kernel ring buffer and stores them to disk. Afterwards, the root filesystem is remounted read-only and the init script exits which triggers a kernel panic and an automated reboot.

4.3.2 Measurements

The machine was booted 101 times and the measurement values of the first run were discarded to get 100 comparable results. The timestamp files were converted into comma separated values using a small python script kernel/data/csv.py. The CSV output was imported into an OpenOffice.org spreadsheet for conversion of cycles into timespans and later visualization. The timespan values were corrected by subtracting the timespan measured between two consecutive measurement points TS_STR and TS_TSC which are executed at the start of each run and do not have any commands inbetween. The resulting time difference expresses the time needed by a single pipeline flush and a read operation to the timestamp counter. Afterwards, negative values were flattened to zero, because there are no negative timespans.

4.3.3 Discussion

Table A.3 and figure B.3 show that most parts of the kernel initialization finish very quickly and only a few routines significantly contribute to the average kernel startup time of 5720ms on the desktop system. The most important routines from init/main.c of the kernel bootup code which consume at least 100µs will be discussed here in detail.

setup_arch is an architecture dependent function which calls the correspondent function from the arch directory and spends about 40.695ms setting up RAM, paging, ACPI, I/O space and probing system, VGA and adapter ROMs.
Routine `sort_main_extable` only needs 650µs to sort the kernel’s built-in exception table, except on architectures alpha, ia64, sparc and v850 where the callee `sort_extable` is empty and should not produce any significant delay.

The `pidhash_init` function needs 118µs to allocate memory for the process identificator hash table and creates a linked list. The size of the table is scaled according to the amount of memory in the machine. The test machine was supplied with 1 gigabyte of RAM which resulted in a pid hash table consisting of 4096 slots. Reducing the RAM size to 512 megabytes instead of 1 gigabyte would have halved the amount of allocated pid hash memory and might help reducing allocation time and the time consumed by a loop which initializes a list head for each hash table entry.

Routine `time_init` runs for 50.142ms. It reads the CMOS real time clock (RTC) by calling the `get_cmos_time` function and afterwards function `pit_calibrate_tsc` determines the CPU frequency by measuring how many ticks the timestamp counter increases in a timespan of 50.00077ms. The programmable interrupt timer (PIT) is used as a time basis, because the initialization of one PIT channel lasts exactly 50.00077ms as stated in [1] chapter 6.1.2. Function `time_init` also prints the CPU frequency and the timekeeping mode selected by the kernel. The remaining 142µs are obviously consumed by the code around the PIT channel initialization code, but the PIT calibration itself takes 99.7% of the `time_init` execution time.

The `console_init` routine needs 1.679ms to run the architecture dependent console initialization code, maybe this delay could be eliminated if there is really no need for any console output. The next routine which consumes more than 100µs is `vfs_cache_init_early` with an average delay of 4.493ms. It initializes directory and inode hash tables by allocating memory for them. The size of the hash tables depends on the total amount of available memory and using less memory could reduce the delay. A time/size tradeoff must be found, because one might run into other problems if there is too little memory available.

The architecture dependent function `mem_init` fills one memory page with zeroes and calls the function `free_all_bootmem`, which frees all boot memory pages. Then it registers memory areas for `/proc/kcore` and finally prints the amount of free and reserved memory. The average execution time of this routine is 10.267ms and this looks like another function whose runtime depends on the total amount of available memory.

`kmem_cache_init` initializes the kernel internal caches for `kmalloc` and registers a CPU startup notifier callback routine to know when a new CPU goes up or down. This routine does not require further investigation, because it is very fast and finished after an average delay of 119µs.

A very time-consuming function instead is `calibrate_delay`, whose average execution time is 77.548ms. It estimates the number of loops per jiffy switch, by calling the delay routine and reading the timestamp counter several times while waiting for a jiffy switch. This delay can be reduced by presetting the loops per jiffy (lpj) value on the kernel command line which completely skips the delay loop calibration and speeds up the boot process.

The `check_bugs` routine runs 78µs and determines which known bugs apply to the target architecture. This could be accelerated by letting the kernel detect the target device bugs in advance and compiling the results into the kernel.

The complete initialization of the ACPI subsystem in function `acpi_early_init` is omitted when the `CONFIG ACPI` kernel configuration option is disabled. If ACPI is definitely not needed, one can
speed up the boot process by at least 6ms, but else it would not be worthwhile to give away the ACPI feature for a speed-up of 6ms.

Function `smp_prepare_cpus`, which has an average delay of 96.54ms on the test system, contains a very time-consuming part for x86-compatible architectures. It contains the `setup_boot_APIC_clock` function which calls the advanced programmable interrupt controller (APIC) bus clock calibration function `calibrate_APIC_clock` for the boot CPU. Calibration times out after a maximum `TICK_COUNT` of 100,000,000 ticks. The APIC bus clock calibration is the third clock calibration in the kernel boot sequence which consumes more than 50ms, but the source code states that jiffies and the timer IRQ could not be used to calibrate the APIC bus clock. If this issue could be resolved, the APIC calibration might be done in a faster way. Additionally, it is always recommended to disable SMP in the kernel configuration on single processor machines in order to disable any SMP related code and achieve a faster bootup.

In our SMP enabled kernel, the `smp_init` routine needs 40.311ms to bring up the CPU of a uniprocessor machine, but if the kernel had been configured in uniprocessor mode, this function would behave differently and initialize the local APIC instead.

At this point CPU and memory management are up, but devices and drivers have not been initialized yet. The most time-consuming part of the kernel bootup which needs more than 4 seconds can be found in function `do_basic_setup` which does the main work. All driver and network initialization routines are called therein. This long delay may be caused by timeouts in the initialization routines of the different drivers and might depend on external devices and their firmware which have to react on a reset request triggered by the corresponding initialization routine. When trying to speed up the kernel boot sequence, one should concentrate on this routine and have a look into all enabled driver initialization routines.

The `prepare_namespace` decides what to mount where. It mounts the root filesystem and loads ramdisks. The test root filesystem was located on an IDE harddisk. So the reason for a delay of 37ms might be the slow harddisk access involving the movement of mechanical components.

Function `ts_free_initmem` fills the kernel initialization memory with character `0xCC` to prevent information leaks and then frees the initialization memory pages of an approximate size of 200 kilobytes. The initdata memory is overwritten with character `0xBA` but not freed. Clearing the init memory areas and printing an informational message result in a total delay of about 200µs.

The system call `sys_open` on device `/dev/console` had an average delay of 32.778ms, so one should disable the initial console if startup can be done without a console.

Printing the profiling timestamps to the console and into the kernel ringbuffer lasts only 1ms, but this is not part of a regular kernel boot process anyway and can therefore be ignored.

The `init_executed` timespan expresses the time between execution of init by the kernel and the executing of a `get-tsc` binary which reads the timestamp counter. The `get-tsc` binary is started in the first line of the init script which is run by the bash interpreter. Therefore the timespan also includes loading a bash shell and libc6 which the shell is linked against. To minimize this delay, one could use a tiny C library instead of libc6 and a tiny shell as provided by ash or busybox.

The previous paragraphs have shown that there is still potential for optimization in some kernel routines, but this can be a complicated and time-consuming task, because the linux kernel is rather complex and things might break if they are simplified or just dropped without adequate preliminary considerations. Optimization must be done step-by-step by developing and testing small patches to the initialization code.
of the kernel and especially the driver initialization code which accounts for about 70% of the kernel boot delay. A total kernel boot time of about 6s is very short in comparison to the service startup time which is handled in the following chapter, but nevertheless 6s can still be a long time when waiting on a cellphone or a digital video recorder to get ready for operation.

The above experiments have also shown that claims [13] to boot an RTLinux kernel in 200ms seem rather unrealistic. First, as we have seen most BIOSes already consume several seconds until any kernel image is loaded after reset, so these 200ms must refer to systems which do not have any BIOS. Writing applications for such systems might be much more difficult and expensive than writing applications for standard systems. Second, a standard featured linux kernel needs about 6s for bootup on a desktop machine, which is 20 times as much as the time claimed by FSMLabs for the quick-boot RTLinux-Pro. In order to boot in 200ms, the boot time of a standard linux kernel would have to be reduced by at least 95%. This is nearly an impossible task, because all time savings observed in this research project were below 70%.

4.4 Init

When the kernel has loaded all drivers and initialized all necessary hardware, services have to be started. The conventional approach implemented in SystemV init is to start one service after another. SystemV init can still be found in most linux distributions, but there exist new approaches like InitNG to start services in parallel. This section discusses characteristics of both approaches and examines how worst-case situations like a dysfunctional network connection influence the startup time when running either SystemV init or InitNG.

4.4.1 Methodology and Setup

A modular linux kernel was used to boot the desktop system. The kernel configuration can be found in file init/install/kernel/config-2.6.14.2.modular.

Bootchartd [7] is used to visualize when userspace services are started and how much time each service consumes. The original bootchartd script waits for mingetty, agetty, rungetty or getty to appear and then gives the getty process 5 seconds to start. The bootchartd script has been modified init/install/boothard/sbin/boothard.default-runlevel to stop profiling as soon as the KDE Login Manager kdm_greet appears, since the original script did not wait for the graphical login screen. This modified script was installed to/sbin/boothard which was passed as the init= argument to the kernel command line in the bootloader configuration.

After installing bootchartd, auto rendering of the boot charts was enabled in file /etc/bootchartd.conf, so that the resulting images could be grabbed from the /var/log directory:

AUTO_RENDER="yes"

4.4.1.1 SystemV Init Setup

The SystemV init is already pre-installed on most linux distributions and/sbin/boothard executes /sbin/init by default, so there is no need for any special setup.
4.4.1.2 InitNG Setup

In order to use InitNG with bootchart, the `/sbin/init` binary has to be replaced with the `/sbin/initng` binary. To get comparable results, InitNG must be configured to start the same services as the standard SystemV init system. This is best done by copying `/etc/initng/default.runlevel` to `/etc/initng/fake-default.runlevel` and commenting out all services with a hash sign. Then the fake mode of initng can be used to test the startup of single services. This way, all necessary services can be added successively without rebooting the machine. When all services and dependencies are in place, the `fake-default.runlevel` file can be copied back to `/etc/initng/default.runlevel`. Then the machine has to be rebooted several times in order to fix the remaining problems with the InitNG service startup. The InitNG configuration used for the Debian GNU/Linux test system can be found in directory `.init/install/initng`.

4.4.2 Measurements

After SystemV init and InitNG had been installed, the AMD Athlon64 desktop machine was rebooted once for each of the following scenarios. When the KDE login screen appeared, the bootchart figure was copied into a subdirectory of `init/data/amd64-asus-a8v/2.6.14.2-modular/default-runlevel`. The first bootchart was created using SystemV init with the network plugged and working. The second bootchart was also created using SystemV init, but network and internet access were made unreachable, which should simulate a network fault. The third and fourth bootchart were created booting InitNG instead of SystemV init, with the two abovementioned network scenarios repeated. Figures B.4 to B.7 show the results.

4.4.3 Discussion

As shown in figure B.4, SystemV init with a working network consumed about 37s for services startup when subtracting 4s kernel startup time which comes before bootchart. The kernel startup time produces a white space at the left border of the CPU utilization bar, because uptime is counted from kernel start, but bootchart is started after the kernel has finished booting. The CPU and disk utilization bar show that neither CPU usage nor harddisk access constitute a bottleneck, because there are white spaces in the diagram where both CPU and I/O are idle. But the serialization of processes produces a staircase effect in the service startup diagram and leads to a long startup delay, because services are blindly started one after another as determined by the order of numbers in the filenames.

Figure B.5 shows that the staircase effect is even intensified when the network connection is cut. The delay is prolonged for services which wait for an answer over the dysfunctional network. This especially affects the `mount` command which tries to mount shared directories and the `ntpd` command which tries to contact a time server on the internet. Both services run into a timeout while the boot process is completely blocked. With SystemV init, the network failure increases the total service startup time by 127s which is 340% above the average-case SystemV init delay.

As shown in figure B.6, the switch to InitNG drastically reduced the staircase effect and lead to a very short service startup time of about 22s (without kernel startup), because services independent from each other could be started in parallel. The right bottom corner of the diagram shows about 16 services starting during one second. Unfortunately, the staircase effect will never disappear completely, because
there will always be certain dependencies between services. An example would be the root filesystem which must be mounted before any programs can be run from there.

The use of InitNG almost nullified the additional delay caused by a dysfunctional network, because other services could use CPU, main memory and harddisk while a few daemons were waiting for an answer on the network and finally timed out. Figure B.7 proves that the InitNG boot process was delayed by only 2s when the network became unavailable. This time, the delay of the worst-case InitNG scenario was only 9% above the average-case InitNG scenario which is much better than the abovementioned SystemV init scenarios.

The conclusion is to always use InitNG or similar utilities which are able to run services in parallel, because they drastically reduce both average-case and worst-case boot delay. Nevertheless, a parallel startup has a few drawbacks like the difficulty to format the output of the different services which will be mixed on the console. A usual solution is to suppress the output completely and just print a percentual progress bar, but sometimes output is needed for debugging purposes. This problem can be resolved by tagging each output line with the corresponding service name where the message came from, or by writing the output from each service to a separate logfile.

James Hunt published some reflections [14] on using `make -j` to start services in parallel, but when comparing this technology with InitNG, several problems became apparent: First, the output of make’s child processes cannot be controlled easily, but maybe a logging wrapper around all child processes could solve this problem. Second, a makefile representing all real, virtual and conditional service dependencies has to be written or there must be a tool which generates such a makefile. Third, the same makefile as used for startup cannot be reused for shutdown. And furthermore, `make` is not able to monitor services or restart them if they terminate unexpectedly. Therefore, it seems that special tools like InitNG are a more reliable method to startup, monitor and shutdown services in parallel.
5 How To Speed Up The Boot Process

Tim Bird [4] lists the following basic methods to speed up the boot process: do it faster, do it in parallel, do it later or don’t do it at all. To do something faster means to clean up and improve the code of BIOS, bootloader, libraries or software. Do it in parallel means to increase the average utilization of the system. To do something later has a meaning similar to don’t do it at all, because if something is done later, it will be obviously deferred and not be available directly after the boot process has finished. This section will present different methods to speed up the boot process and will classify them into the above categories.

5.1 Optimize the BIOS configuration

A good BIOS configuration can shorten BIOS startup times of desktop systems by about 50% of the worst case time when using an AMI BIOS. Unfortunately, this does not apply to embedded systems, because most embedded BIOSes are very specific to the target architecture and have fewer user settings.

5.2 LinuxBIOS

Table A.2 shows that boot times could be shortened by 5 seconds. This was achieved by totally getting rid of the original factory BIOS and writing an optimized LinuxBIOS image into the BIOS flash chip. LinuxBIOS is open source and can therefore be optimized to execute neither more nor less code than necessary. It only initializes hardware needed to continue the boot process and follows the basic rules do it faster and don’t do it at all.

5.3 Tiny Bootloader

The change from VIA factory BIOS to LinuxBIOS was directly connected with a change from bootloader GRUB (leftmost bar in figure B.2) to bootloader FILO (second bar in figure B.2), because FILO is one of a few common payloads for LinuxBIOS and GRUB might be too large as a LinuxBIOS payload. So the use of LinuxBIOS might not be the only reason for the observed speedup. The use of a light bootloader can also help speed up the boot process, not only by disabling console output and user interaction, but also by dropping unnecessary drivers and code.

5.4 Predefine Loops per Jiffy

The kernel documentation states that predefining the loops per jiffy value saves up to 250ms during each kernel startup by omitting the delay loop calibration. The $lpj$ value has to be manually determined for each target platform by reading the kernel messages and then the fixed value must be passed at the kernel
command line. This is a good example for the basic rule *not to do it at all*, because calibration is done only once.

### 5.5 Disable Console

The Linux kernel writes a lot of messages to the output console by default. Adding the *quiet* option to the kernel command line disables almost all kernel initialization messages and can therefore be sorted into category *do it faster*. It is also very important to disable any null-modem serial console in BIOS, bootloader and kernel, because the serial console slows down the boot process. This can be seen in the second and third scenario of figure B.2, where one version of FILO with serial console support and another one without serial console support have been compared. Nevertheless, the difference between the boot processes where the BIOS and bootloader console was either enabled or disabled was only 200 milliseconds. Disabling the console can help to speed up the boot process, but is only a small contribution in comparison to replacing the whole BIOS or bootloader.

### 5.6 Replace Init with InitNG

The above experiments have shown that InitNG can reduce service startup time of a common Linux system by about 40%. InitNG is based on the principle *do it in parallel* and makes far better use of available resources than a conventional SystemV init system, because while one process is waiting for something, all other active services can continue starting. The two upper bars of figure B.6 illustrate that there is almost no time during an InitNG startup sequence where CPU or I/O system are completely idle. In contrast, the service startup sequence of SystemV init as shown in figure B.4 contains several idle times where processes are blocked or simply sleeping and stall the whole boot process.

### 5.7 Storage Medium

The two rightmost bars of figure B.2 show that the spin-up of the harddisk consumes a lot of additional time until the kernel can be loaded. In order to save time, it is recommended to use a flash ROM or a fast network to load the kernel.

### 5.8 Lighter Libraries and Software

The last row of table A.3 revealed that on a standard Debian GNU/Linux desktop system, there was still a one-second delay between the execution of the init executable by the kernel and the execution of a binary in the first line of the init shell script. One could try to reduce this delay by installing lighter versions of the C libraries and a lighter shell, but replacing the C libraries will require rebuilding the system from scratch, because usually a lot of programs are dynamically linked against these libraries.
6 Further work

Porting LinuxBIOS

The above experiments have shown that LinuxBIOS boots faster than most factory BIOSes. It would be a great contribution to the linux community, if LinuxBIOS was ported to an architecture which is not supported yet. The people on the LinuxBIOS mailing list can help porting LinuxBIOS and given the target chipsets they can also tell if porting to a certain architecture would be possible at all. Two short guides about porting LinuxBIOS are mentioned on the mailing list, but there is no formal porting documentation yet, so it would be very useful to write a comprehensive and abstract documentation about porting LinuxBIOS to new architectures as well.

Graphics initialization without factory VGA BIOS

During the experiments it was observed that the initialization of the on-board VGA graphics adapter on the VIA EPIA-ML6000EA takes a significant part (about 3s) of the LinuxBIOS startup time until the first screen output appears. It should be investigated how the graphics adapter is initialized and if there are ways to boot LinuxBIOS and use graphical output without prepending a proprietary VGA BIOS image to the LinuxBIOS image.

Detailed Kernel Analysis

Though the kernel measurements have shown that a standard kernel only needs about 6s to start, it would be useful to do a profound kernel analysis which concentrates on the do_basic_setup function in order to find time lags in single drivers. It should be investigated if there are opportunities to initialize kernel drivers in parallel, in a similar way like InitNG does with user space services. The kernel analysis should include a comparison between single processor and multi processor machines, because several driver initialization routines could be handled by different CPUs at the same time.

Booting from network or flash

Since harddisk drives are very slow and their spin-up significantly delays the boot process, booting from harddisk should be compared with alternative boot methods like booting from a fast network or flash memory. When flash is used for the root filesystem, the execute in place (XIP) feature could be evaluated.
7 Summary

The main goal of this project was to analyse the linux boot process and to find out the most significant parts when it comes to reducing boot time.

Experiments have shown that BIOS (7.6s to 48s) and service startup (22s to 164s) take most of the boot time, while the linux kernel itself takes only about 6s. Figures B.1 and B.2 show that the factory BIOS on embedded systems boots generally faster (5s to 12s) than on standard desktop systems (19s to 48s). Table B.1 also made clear that disabling unused BIOS features reduces startup time in most cases.

There is a general problem with optimizing boot speed. If a system shall boot faster, it must be customized. Flexible and universal startup scripts which cover several cases can still be found on most linux distributions. These scripts are very portable and run well on many architectures without any change, but with the cost of longer boot times.

Measurements have shown that it is really possible to speed up the linux boot process by about 40% with some efforts. The most important contributions come from LinuxBIOS and InitNG. It is important to optimize and port LinuxBIOS to new architectures as well as to distribute InitNG with standard linux distributions.
## A Tables

### Delay of BIOS & Bootloader

<table>
<thead>
<tr>
<th>BIOS Configuration</th>
<th>Minimum (s)</th>
<th>Arithmetic Mean (s)</th>
<th>Maximum (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case</td>
<td>47.50</td>
<td>47.63</td>
<td>47.74</td>
</tr>
<tr>
<td>MIDI disabled</td>
<td>47.52</td>
<td>47.56</td>
<td>47.60</td>
</tr>
<tr>
<td>LPT disabled</td>
<td>47.46</td>
<td>47.57</td>
<td>47.77</td>
</tr>
<tr>
<td>COM2 disabled</td>
<td>47.49</td>
<td>47.54</td>
<td>47.57</td>
</tr>
<tr>
<td>COM1 disabled</td>
<td>47.54</td>
<td>47.56</td>
<td>47.61</td>
</tr>
<tr>
<td>Firewire disabled</td>
<td>47.55</td>
<td>47.58</td>
<td>47.61</td>
</tr>
<tr>
<td>LAN Boot disabled</td>
<td>47.31</td>
<td>47.44</td>
<td>47.60</td>
</tr>
<tr>
<td>LAN disabled</td>
<td>47.33</td>
<td>47.38</td>
<td>47.59</td>
</tr>
<tr>
<td>RAID disabled</td>
<td>37.14</td>
<td>37.19</td>
<td>37.24</td>
</tr>
<tr>
<td>SATA disabled</td>
<td>33.49</td>
<td>33.51</td>
<td>33.53</td>
</tr>
<tr>
<td>Audio disabled</td>
<td>33.49</td>
<td>33.52</td>
<td>33.56</td>
</tr>
<tr>
<td>INT19 ROMs disabled</td>
<td>33.50</td>
<td>33.53</td>
<td>33.56</td>
</tr>
<tr>
<td>&quot;Del&quot; Message disabled</td>
<td>33.50</td>
<td>33.53</td>
<td>33.57</td>
</tr>
<tr>
<td>Quick Boot enabled</td>
<td>21.79</td>
<td>21.82</td>
<td>21.86</td>
</tr>
<tr>
<td>Floppy Boot disabled</td>
<td>20.69</td>
<td>20.74</td>
<td>20.79</td>
</tr>
<tr>
<td>CD-ROM Boot disabled</td>
<td>20.48</td>
<td>20.62</td>
<td>20.77</td>
</tr>
<tr>
<td>Floppy disabled</td>
<td>26.46</td>
<td>26.47</td>
<td>26.51</td>
</tr>
<tr>
<td>Floppy enabled</td>
<td>20.48</td>
<td>20.58</td>
<td>20.75</td>
</tr>
<tr>
<td>CD-ROM disabled</td>
<td>20.50</td>
<td>20.57</td>
<td>20.74</td>
</tr>
<tr>
<td>USB disabled</td>
<td>18.90</td>
<td>18.93</td>
<td>18.97</td>
</tr>
<tr>
<td>MDA resources disabled</td>
<td>18.88</td>
<td>18.92</td>
<td>18.95</td>
</tr>
<tr>
<td>ACPI/APIC disabled</td>
<td>18.86</td>
<td>18.92</td>
<td>18.96</td>
</tr>
</tbody>
</table>

Table A.1: AMI BIOS on Asus A8V Deluxe Board with Athlon64 3000+ at 1800 Mhz

<table>
<thead>
<tr>
<th>BIOS</th>
<th>Bootloader</th>
<th>HDD</th>
<th>Minimum (s)</th>
<th>Arithmetic Mean (s)</th>
<th>Maximum (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Award BIOS</td>
<td>GRUB</td>
<td>running</td>
<td>12.77</td>
<td>12.79</td>
<td>12.80</td>
</tr>
<tr>
<td>Linux BIOS</td>
<td>standard FILO</td>
<td>running</td>
<td>7.79</td>
<td>7.80</td>
<td>7.80</td>
</tr>
<tr>
<td>Linux BIOS</td>
<td>optimized FILO</td>
<td>running</td>
<td>7.59</td>
<td>7.60</td>
<td>7.60</td>
</tr>
<tr>
<td>Linux BIOS</td>
<td>optimized FILO</td>
<td>spin-up</td>
<td>11.29</td>
<td>11.37</td>
<td>11.41</td>
</tr>
</tbody>
</table>

Table A.2: Award and LinuxBIOS on VIA EPIA-ML6000EA Board
## Delay of Kernel Startup

<table>
<thead>
<tr>
<th>Kernel Action</th>
<th>Arithmetic Mean (cycles)</th>
<th>Arithmetic Mean (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts_lock_kernel</td>
<td>309.55</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_page_address</td>
<td>0.00</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_kern_notice</td>
<td>2060.32</td>
<td>0.001</td>
</tr>
<tr>
<td>ts_linux_banner</td>
<td>4996.41</td>
<td>0.003</td>
</tr>
<tr>
<td>ts_setup_arch</td>
<td>73250678.32</td>
<td>40.695</td>
</tr>
<tr>
<td>ts_setup_per_cpu_areas</td>
<td>74765.18</td>
<td>0.042</td>
</tr>
<tr>
<td>ts_smp_prepare_boot_cpu</td>
<td>446.74</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_sched_init</td>
<td>65804.44</td>
<td>0.037</td>
</tr>
<tr>
<td>ts_preempt_disable</td>
<td>0.00</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_build_all_zonelists</td>
<td>4837.71</td>
<td>0.003</td>
</tr>
<tr>
<td>ts_page_alloc_init</td>
<td>82.03</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_kernel_commandline</td>
<td>3408.90</td>
<td>0.002</td>
</tr>
<tr>
<td>ts_parse_early_param</td>
<td>7379.62</td>
<td>0.004</td>
</tr>
<tr>
<td>ts_parse_args</td>
<td>32354.03</td>
<td>0.018</td>
</tr>
<tr>
<td>ts_sort_main_extable</td>
<td>1170631.46</td>
<td>0.650</td>
</tr>
<tr>
<td>ts_trap_init</td>
<td>12485.00</td>
<td>0.007</td>
</tr>
<tr>
<td>ts_rcu_init</td>
<td>2712.83</td>
<td>0.002</td>
</tr>
<tr>
<td>ts_init_IRQ</td>
<td>110931.75</td>
<td>0.062</td>
</tr>
<tr>
<td>ts_pidhash_init</td>
<td>212350.69</td>
<td>0.118</td>
</tr>
<tr>
<td>ts_init_timers</td>
<td>3270.36</td>
<td>0.002</td>
</tr>
<tr>
<td>ts_softirq_init</td>
<td>292.33</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_time_init</td>
<td>90255065.96</td>
<td>50.142</td>
</tr>
<tr>
<td>ts_console_init</td>
<td>3021497.00</td>
<td>1.679</td>
</tr>
<tr>
<td>ts_panicLater</td>
<td>132.35</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_profile_init</td>
<td>256.25</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_local_irq_enable</td>
<td>48855.19</td>
<td>0.027</td>
</tr>
<tr>
<td>ts_blk_dev_initrd</td>
<td>319.50</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_vfs_caches</td>
<td>8086746.21</td>
<td>4.493</td>
</tr>
<tr>
<td>ts_mem_init</td>
<td>18481191.46</td>
<td>10.267</td>
</tr>
<tr>
<td>ts_kmem_cache_init</td>
<td>213578.21</td>
<td>0.119</td>
</tr>
<tr>
<td>ts_setup_per_cpu_pageset</td>
<td>2856.62</td>
<td>0.002</td>
</tr>
<tr>
<td>ts_numa_policy_init</td>
<td>33332.15</td>
<td>0.019</td>
</tr>
<tr>
<td>ts_late_time_init</td>
<td>145.49</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_calibrate_delay</td>
<td>139585864.00</td>
<td>77.548</td>
</tr>
<tr>
<td>ts_pidmap_init</td>
<td>8764.52</td>
<td>0.005</td>
</tr>
<tr>
<td>ts_pgtable_cache_init</td>
<td>133.78</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_prio_tree_init</td>
<td>1248.04</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Table A.3: Kernel 2.6.14.2 on Asus A8V Deluxe Board with Athlon64 3000+ at 1800 Mhz
<table>
<thead>
<tr>
<th>Kernel Action</th>
<th>Arithmetic Mean (cycles)</th>
<th>Arithmetic Mean (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ts_anon_vma_init</td>
<td>7165.61</td>
<td>0.004</td>
</tr>
<tr>
<td>ts_efi_enter_virtual_mode</td>
<td>0.00</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_fork_init</td>
<td>5978.43</td>
<td>0.003</td>
</tr>
<tr>
<td>ts_proc_caches</td>
<td>31582.09</td>
<td>0.018</td>
</tr>
<tr>
<td>ts_buffer_init</td>
<td>7603.11</td>
<td>0.004</td>
</tr>
<tr>
<td>ts_unnamed_dev_init</td>
<td>5546.48</td>
<td>0.003</td>
</tr>
<tr>
<td>ts_key_init</td>
<td>106.65</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_security_init</td>
<td>0.00</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_vfs_caches_init</td>
<td>154052.57</td>
<td>0.086</td>
</tr>
<tr>
<td>ts_radix_tree</td>
<td>6210.89</td>
<td>0.003</td>
</tr>
<tr>
<td>ts_signals_init</td>
<td>6817.15</td>
<td>0.004</td>
</tr>
<tr>
<td>ts_page_writeback_init</td>
<td>2613.07</td>
<td>0.001</td>
</tr>
<tr>
<td>ts_proc_root_init</td>
<td>65670.74</td>
<td>0.036</td>
</tr>
<tr>
<td>ts_cpuset_init</td>
<td>98.02</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_check_bugs</td>
<td>141103.33</td>
<td>0.078</td>
</tr>
<tr>
<td>ts_acpi_early_init</td>
<td>11558884.79</td>
<td>6.422</td>
</tr>
<tr>
<td>ts_initRoutine</td>
<td>39349.23</td>
<td>0.022</td>
</tr>
<tr>
<td>ts_lock_kernel2</td>
<td>142.39</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_set_cpus_allowed</td>
<td>1174.31</td>
<td>0.001</td>
</tr>
<tr>
<td>ts_smp_prepare_cpus</td>
<td>173771388.33</td>
<td>96.540</td>
</tr>
<tr>
<td>ts_do_pre_smp_initcalls</td>
<td>56642.14</td>
<td>0.031</td>
</tr>
<tr>
<td>ts_fixup_cpu_present_map</td>
<td>159.35</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_smp_init</td>
<td>72559941.29</td>
<td>40.311</td>
</tr>
<tr>
<td>ts_sched_init</td>
<td>87713.76</td>
<td>0.049</td>
</tr>
<tr>
<td>ts_cpuset_init_smp</td>
<td>0.00</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_populate_rootfs</td>
<td>149705.76</td>
<td>0.083</td>
</tr>
<tr>
<td>ts_do_basic_setup</td>
<td>7641696704.08</td>
<td>4245.387</td>
</tr>
<tr>
<td>ts_set Ramdisk_execute_command</td>
<td>134.23</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_prepare_namespace</td>
<td>67202551.36</td>
<td>37.335</td>
</tr>
<tr>
<td>ts_free_initmem</td>
<td>355971.94</td>
<td>0.198</td>
</tr>
<tr>
<td>ts_unlock_kernel</td>
<td>136.79</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_set_system_state</td>
<td>94.26</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_numa_default_policy2</td>
<td>1880.11</td>
<td>0.001</td>
</tr>
<tr>
<td>ts_open_console</td>
<td>59000158.73</td>
<td>32.778</td>
</tr>
<tr>
<td>ts_sys_dup_fdzero1</td>
<td>1835.80</td>
<td>0.001</td>
</tr>
<tr>
<td>ts_sys_dup_fdzero2</td>
<td>182.06</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_run_init_ramdisk</td>
<td>201.46</td>
<td>0.000</td>
</tr>
<tr>
<td>ts_print_timestamps</td>
<td>1978322.53</td>
<td>1.099</td>
</tr>
<tr>
<td>ts_init_executed</td>
<td>1933212286.85</td>
<td>1074.007</td>
</tr>
</tbody>
</table>

Table A.3: Kernel 2.6.14.2 on Asus A8V Deluxe Board with Athlon64 3000+ at 1800 Mhz
Figure B.1: AMI BIOS on Asus A8V Deluxe Board with Athlon64 3000+ at 1800 Mhz

Figure B.2: Award and LinuxBIOS on VIA EPIA-ML6000EA Board
Figure B.3: Kernel 2.6.14.2 on Asus A8V Deluxe Board with Athlon64 3000+ at 1800 Mhz
Figure B.4: Service boot process with SystemV init and working network
Figure B.5: Service boot process with SystemV init and unreachable network
Figure B.6: Service boot process with InitNG and working network
Figure B.7: Service boot process with InitNG and unreachable network
Bibliography


   http://tree.celinuxforum.org/CelfPubWiki/KernelFunctionTrace

   http://tree.celinuxforum.org/CelfPubWiki/KernelFunctionTrace#
   head-afffe81c6dad8cc96b3ce448af31b5dc40135548

   http://tree.celinuxforum.org/CelfPubWiki/BootupTimeResources?
   action=AttachFile&do=get&target=ReducingStartupTime\_v0.8.pdf

[5] Opersys inc.: Linux Trace Toolkit
   http://www.opersys.com/LTT/


[7] Ziga Mahkovec: Bootchart - Visualize service startup
   http://www.bootchart.org/

[8] Jimmy Wennlund: InitNG - A SystemV init replacement which runs services in parallel
   http://initng.thinktux.net/

[9] LinuxBIOS: An open source BIOS replacement
   http://www.linuxbios.org/

[10] FILO: A tiny bootloader which acts as a payload for LinuxBIOS
    http://www.linuxbios.org/index.php/FILO

[11] VIA: EPIA-M BIOS 1.13 - Contains a VGA BIOS image known to work with LinuxBIOS
    http://www.via.com.tw/download/mainboards/1/0/i0100113.bin

[12] Award BIOS Editor
    http://awdbedit.sourceforge.net/

    http://www.fsmlabs.com/rtlinuxpro.html