dotSource®

Software- und Hardwaretest

Minimal Linux

Abgabetermin: Gera, den 16.11.2018

Studienarbeit

von:	[redacted]
Matrikelnummer:	[redacted]
Duale Hochschule:	Duale Hochschule Gera-Eisenach Weg der Freundschaft 4a 07546 Gera
Studienbereich:	Technik
Studiengang:	Praktische Informatik
Kurs:	PI 16
Ausbildungsstätte:	dotSource GmbH Goethestraße 1 07743 Jena
Betreuer DHGE:	[redacted]

Inhaltsverzeichnis

Abbildungsverzeichnis

Auflistungen

Abkürzungsverzeichnis

BIOS Basic Input/Output System

CMOS Complementary metal-oxide-semiconductor

DHGE Duale Hochschule Gera-Eisenach

EFI Extensible Firmware Interface

FAT File-Allocation-Table

GDT Global Descriptor Table

GPT GUID-Partition-Table

IDT Interrupt Descriptor Table

MBR Master Boot Record

MLL Minimal-Linux-Live

POST Power-on self test

UDF Universal Disk Format

UEFI Unified Extensible Firmware Interface

1 Einleitung

Beinahe tägliche schreitet die Technik der Gegenwart voran und Endnutzer sind oft überfordert von der Auswahl. Bei diesem Problem helfen Benchmarks und andere Ratingverfahren. Durch diese bleibt dem Nutzer die mühselige Auseinandersetzung mit den technischen Details erspart und die Parameter zum Kauf neuer Produkte reduzieren sich auf Preis und Ratingpoints. Doch was steckt hinter diesen Verfahren zum Messen der Leistungsfähigkeit von modernen Computern und Laptops und wie repräsentativ sind diese?
Microsoft bietet von Haus aus eine Leistungsindexermittlung an, die im laufenden System ausgeführt wird und dabei die aktuelle Hardware testet und eine Punktzahl ausgibt, die in einer Liste mit anderen ähnlichen Konfigurationen anderer esitzer verglichen werden kann. Ferner gibt es Anwendungssoftware, die mitunter detailiertere Ergbnisse präsentieren. Doch wo der Endnutzer den Zahlen vertraut, sollte die Fachkraft die Zahlen und deren Erhebungsart hinterfragen.

1.1 Problemstellung

Im Rahmen des Studium der praktischen Informatik, müssen die Studenten der Dualen Hochshule Gera[1] ein Praktikum zum Thema Computer-Hardware absolvieren. In diesem Praktikum werden Workstations montiert und grundsätzliche Hardware- und Leistungstests durchgeführt, so dass die Teilnehmer ein Gefühl für die Montage entwickeln und sich praxisnah mit den Kennlinien der einzelnen Schnittstellen und Konfigurationen auseinandersetzen können. Darüber hinaus werden Teile wie Festplatten über verschiedene Controller mit dem System verbunden und RAM-Module ausgetauscht, um so die Unterschiede in der Leistung zu verdeutlichen. Bisher wurden dafür Programme wie zum Beispiel CPU-Z[2] für das Testen der CPU und AIDA64[3] (siehe hierzu Abb. 1.1) für weitreichende Systeminformationen unter anderem aber auch Memory Benchmarks verwendet. Diese Applikationen werden auch im Endnutzerbreich stark genutzt. Sie liefern ein mehr oder minder vergleichbares Ergebnis und helfen zumindest bei der Einschätzung der aktuellen Leistungsfähigkeit und ermöglichen somit die Verbesserung und Feinanpassung der präsenten Hardware. Abseits der genannten Beispiele gibt es eine Vielzahl an weiteren Programmen, die innerhalb des laufenden Systems - also nach dem Start des Betriebssystems - die Messung durchführen. Doch dies bringt diverse Probleme mit sich. Ihnen ist gemein, dass die Erhebung der Daten für die Leistungsbestimmung

[1] im folgenden als DHGE bezeichnet
[2] Link: https://cpuid.com/softwares/cpu-z.html
[3] Link: https://www.aida64.com/

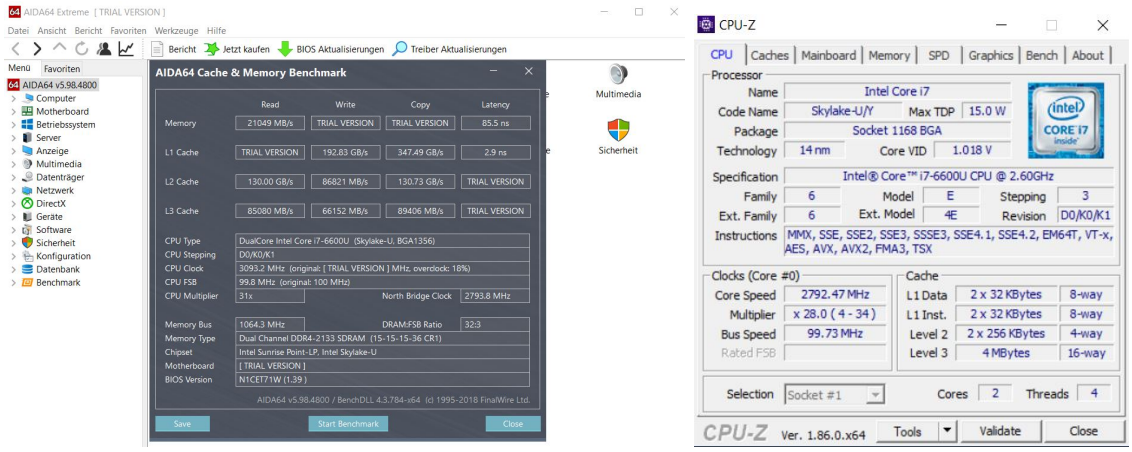

Abbildung 1.1: Systeminformationen u. Benchmark: Aida64 (links) u. CPU-Z (rechts)[4]

teilweise durch Programme des laufenden Systems gestört werden können und somit die Messung mit unter weniger genau oder zumindest nicht exakt ist. Wenngleich ein absolut exaktes Ergebnis nicht oder verhältnismäßig schwierig zu erreichen ist, können die Unterbrechungen und Beeinträchtigungen minimiert werden.
Die Abweichungen haben viele Gründe: Zum einen verwaltet das Betriebssystem diverse Dienste, die immer wieder zahlreiche Zwischenprüfungen oder Aufgaben durchführen. Darüber hinaus können auch manuell installierte Dienste die Messung stören. Zum anderen kann das Betriebsystem auch Mechanismen verwenden, die im täglichen Betrieb die Leistung beschleunigen wie zum Beispiel das Buffering oder die Cache-Methoden. Dennoch würden diese zum Beispiel beim Test der Schreibvorgänge auf einen Datenträger das Ergebnis absolut verfälschen.
Ein Teil der weiter oben beschriebenen Probleme können durch die Nutzung eines Systems vor dem Start des Betriebssystems umgangen werden. Dabei wird ein möglichst minimales Betriebssystem gestartet, dass ausschließlich für den Betrieb notwendige Dienste startet und Treiber lädt. Somit werden die Fehler reduziert und Testprogramme können relevantere Daten liefern. Ein weiterer Vorteil ist das schnelle Starten des Systems, so dass Konfigurationsanpassungen am System einschließlich der Messung schneller durchgeführt werden können.

1.2 Zielstellung

Das Projekt ist in zwei Teile gegliedert: Der erste Teil beschäftigt sich mit der Erstellung der Umgebung, in der das Leistungserhebungswerkzeug ausgeführt wird. Der zweite Teil beschreibt die Erstellung des Messwerkzeugs. Die vorliegende Arbeit beschränkt sich auf den ersten Teil.
Dabei geht es im Detail darum eine Umgebung zu bauen, die zum einen schnell gestartet werden kann und zum anderen möglichst minimal ausgerüstet ist. Als Basis wird ein Linux-System gefordert und als Referenzbeispiel für den Einsatz dienen die Rechner des Praktikums der DHGE. Darüber hinaus soll die Arbeit die Möglichkeit bieten, dem

[4] Quelle: selbst erstellte Grafik

Leser eine Anpassung der Konfiguration vorzunehmen und das Erstellen des Images selbstständig durchzuführen. Die genauen Anforderungen, die sich vor und im Laufe der Arbeit am Projekt ergeben haben, werden im Anhang A.1 zusammengefasst dargestellt und in Kapitel 2.2 genauer ausgeführt.
Es soll zunächst ein grober Überblick an Möglichkeiten zum Erstellen dieses Systems erfolgen. Durch die Auswahl einer Methode kann die Arbeit fokussiert fortgesetzt werden. Bevor die praktische Umsetzung erfolgt, soll durch eine theoretische Betrachtung des Startvorgangs und der damit verbundenen Umstände die Basis für die Entscheidungsfindungen während der Arbeit und für das weitere Vorgehen gelegt werden.
Das abschließende Ziel der Arbeit ist es einen Datenträgerimage zu produzieren, der unter den weiter oben beschriebenen Anforderungen genutzt werden kann. Das System wird an den Laborcomputern der Hochschule getestet und auf einem geeigneten Medium der Hochschule zur Verfügung gestellt werden. Optional ist ein Build-Skript zu übergeben, dass auch die spätere Überarbeitung bzw. Erstellung der Images ermöglicht.
Die vorliegende Arbeit ist für Fachpersonal bestimmt, weshalb eine Basis an Wissen des Bereichs vorausgesetzt wird und somit nicht alle Erklärungen bis in die Tiefe gehen werden.

1.3 Vorgehensweise

Die Studienarbeit ist in vier Kapitel gegliedert: die Einleitung, die Vorbetrachtung und Vorbereitung, die Realisierung und das Fazit.
Nach einer kurzen Einstimmung in das Thema erfolgt die Beschreibung des Problems, sowie die Darlegung des Ziels und der Abgrenzung. Durch das Umreißen der Vorgehensweise wird das Kapitel abgeschlossen.
Im zweiten Kapitel wird zunächst der grundsätzliche Vorgang des Starten eines Computers oder Laptops beschrieben. Dadurch resultiert ein Teil der Anforderungen und ein grober Entwurf der späteren Umgebung. Anschließend werden diverse Anforderungen an die Umgebung genauer betrachtet, um den Rahmen für das Zielsystem enger zu fassen. Abschließend werden alternative Methoden zur Realisierung des Projektes kurz umrissen, wobei die genaue Darlegung der ausgewählten Methode durch den praktischen Teil beschrieben wird.
Die Realisierung beschreibt die Umsetzung des Projektes im Kern und dokumentiert die Vorgehensweise. Dabei wird zunächst am Anfang das Build-Skript von *Minimal-Linux-Live* im Detail diskutiert. Anschließend werden die Anpassungen am Skript und die Überführung in das erweiterte Skript thematisiert. Dabei wird der Umgang mit der Kernelkonfiguration beschrieben und welche Optionen eingeführt wurden um die Anforderungen abzudecken. Den Abschluss bildet die Bearbeitung des Boot-Mediums und die Auswertung der Testlaufes.
Im letzten Teil der Arbeit wird ein Resümee über die Studienarbeit gezogen und eine Auswertung bezüglich der Zielstellung gemacht. Abschließend werden Verbesserungsvorschläge festgehalten, die im Rahmen der Arbeit nicht mehr realisiert werden konnten.

2 Vorbetrachtung und Vorbereitung

Dieses Kapitel beschäftigt sich mit der theoretischen Vorbetrachtung und der Vorbereitung des Projektes. Dabei wird zunächst der generelle Startprozess eines Heimrechners beschrieben. Dadurch wird bereits ein Bild vom Zielsystem gezeichnet. Anschließend werden die Anforderungen an das Zielsystem thematisiert und aufbereitet und führen somit zu einer Spezifizierung des späteren Systems. Im dritte Teil des Kapitels wird der manuelle Ansatz zur Kernelkonfiguration dargelegt, da dies neben der Recherche der Einstiegspunkt in die Studienarbeit war und es wird eine verfügbare Alternative zu *Minimal-Linux-Live* kurz angerissen und begründet, warum diese und andere Vertreter nicht zum Einsatz kamen. Da das Zielsystem Linux als Basis nutzen wird, werden die folgenden Ausführungen stark Linux orientiert sein.

2.1 Der Bootvorgang

Der Bootvorgang steht ganz am Anfang jeder Prozesskette eines Rechners. Bevor überhaupt eine gewünschte Funktion, wie zum Beispiel das Browsen im Internet oder das Verfassen von Dokumenten, genutzt werden kann, muss der Rechner starten. Dabei wird durch das Anschließen einer Stromquelle, wie beim RasberryPi, oder durch das Drücken des Ein-/Ausschalter zunächst das Startsignal gegeben und das Basic Input/Output System (BIOS) - die Firmware des Motherboards - gestartet. Das BIOS ist eine Low-Level-Software, die auf dem Motherboad installiert ist und zwischen der Hardware - insbesondere dem Chipsatz und der CPU - und dem Betriebssystem als Interface arbeitet[1]. Zuerst führt es eine Hardwareerkennung durch und initialisiert diese. Dabei werden die minimal notwendigen Hardwarekomponenten (CPU, RAM und ggf. Grafikkarte) getestet - was auch als Power-on self test (POST) bezeichnet wird - , der Arbeitsspeicher eingebunden und die Hardwareeinstellungen initialisiert[2]. Ab dem jetzigen Zeitpunkt ist der Boot-Modus interessant, wobei zwischen BIOS- und EFI-Modus unterschieden wird. Das Bios stellt die klassische Variante dar und wird heutzutage noch weitreichend genutzt. Extensible Firmware Interface (EFI), bzw. Unified Extensible Firmware Interface (UEFI) als Nachfolger, sind die moderneren und zeitgemäßeren Modi. Dennoch sei angemerkt:"Das alte BIOS wird nicht komplett durch UEFI ersetzt. Dinge wie der POST (Power On Self Test) oder die grundlegende Konfiguration über das BIOS-Setup werden weiterhin über das BIOS geregelt."[3]

[1] Vgl. Koz01.
[2] Vgl. Her06, S. 367.
[3] Vgl. ubu18.

2.1.1 Das BIOS

Das BIOS lädt nun die Einstellungen, wie zum Beispiel die Bootreihenfolge, aus dem Complementary metal-oxide-semiconductor (CMOS) in den Speicher und lädt die Partitionstabelle und den Master Boot Record (MBR) des Mediums. Der MBR umfasst in der Regel das Bootprogramm, dass die Einträge für die Betriebssysteme auf den Partitionen enthält[4]. Das BIOS übergibt die Kontrolle an den Bootmanager. Entsprechend der Auswahl des Systems, sucht das Bootprogramm auf der hinterlegten Partition nach dem Kernel, startet diesen und übergibt die Kontrolle diesen. Alternativ können aufgrund von Platzproblemen im MBR weitere Bootloader von den Partitionen gestartet werden.

2.1.2 UEFI

UEFI besitzt weit mehr Möglichkeiten als BIOS. Im UEFI-Modus kann wesentlich früher als beim BIOS-Modus High-Level-Code ausgeführt werden und die Firmware besitzt eigene Treiber zum Einbinden von Festplatten oder für Netzwerkkarten. Zusätzlich kann die Firmware Partitionstabellen lesen und versteht FAT-Filesysteme[5].
Das Motherboard hat einen mitgelieferten UEFI-Bootmanager und verfügt über diverse Variablen, die durch das System gesetzt werden können. Das folgende Beispiel wurde im Rahmen der Studienarbeit auf einer virtuellen Maschine ausgeführt und soll die Thematik anschaulicher beschreiben. So zeigt die Auflistung 2.1 die UEFI-Einträge, die durch den Befehl *efibootmgr -v* ausgegeben wurden. Hierbei handelt es sich um die Einträge, die zur Zeit im EFI-Bootmanager des Motherboards gespeichert sind.

Auflistung 2.1: Beispiel 'efibootmgr' Ausgabe

```
1 root@debian_sa_minimal_linux:/boot/efi/EFI/debian$ efibootmgr -v
2 BootCurrent: 0002
3 BootOrder: 0000,0001,0002
4 Boot0000* EFI DVD/CDROM PciRoot(0x0)/Pci(0x1,0x1)/Ata(1,0,0)
5 Boot0001* EFI Hard Drive     PciRoot(0x0)/Pci(0xd,0x0)/Sata(0,0,0)
6 Boot0002* EFI Internal Shell  MemoryMapped(11,0x2100000,0x28fffff)/FvFile
    ↪ (7c04a583-9e3e-4f1c-ad65-e05268d0b4d1)
```

Der Eintrag *BootCurrent* gibt den Menüeintrag an, über den aktuell gebootet wurde und die Bootreihenfolge wird durch *BootOrder* beschrieben. Darauf folgen die Menüeinträge, wobei zu erst von einer CD oder DVD gebootet werden kann, danach von einer angeschlossenen Festplatte und zuletzt die interne Shell des Bootmanagers gestartet wird. Wie bereits weiter oben beschrieben, kann bei UEFI wesentlich früher High-Level-Code ausgeführt werden. Bei jedem Start wird das Skript *startup.nsh* - zu sehen in der Auflistung 2.2 - automatisch ausgeführt[6].

Auflistung 2.2: startup.nsh

```
1 root@debian_sa_minimal_linux:/boot/efi$ cat startup.nsh
```

[4] Vgl. Her06, Ebd.
[5] Vgl. Wil14.
[6] Vgl. und weitere Informationen zum Shellscripten für EFI unter int08.

```
2 FS0:
3 \EFI\debian\grubx64.efi
```

Die Ausführung oder Abarbeitung der Datei erinnert stark an MS-DOS Anweisungen: Durch die erste Zeile wird auf die erste Festplatte gewechselt. *FS0* steht für das Diskettenlaufwerk, wenn eine Diskette vor dem Start des PC eingelegt wurde, andernfalls für die erste Festplatte. Der Bootmanager sucht auf dieser nach einer EFI-Partition. Wie die Auflistung 2.3 zeigt, ist in der GPT der Festplatte *sda* ein EFI-System - nämlich */dev/sda1* - hinterlegt.

Auflistung 2.3: Testmaschine: fdisk -l

```
1 root@debian_sa_minimal_linux:/boot/efi/EFI/debian$ fdisk -l
2 Disk /dev/sda: 20 GiB, 21474836480 bytes, 41943040 sectors
3 ...
4 Disklabel type: gpt
5 ...
6 Device     Start    End Sectors Size Type
7 /dev/sda1   2048 1050623 1048576 512M EFI System
8 /dev/sda2  1050624 39843839 38793216 18.5G Linux filesystem
9 /dev/sda3 39843840 41940991 2097152  1G Linux swap
```

Die Partition wurde im laufenden System unter */boot/efi* eingebunden wie die Ausgabe von *mount* in der Ausgabe 2.4 zeigt.

Auflistung 2.4: mount |grep sda1

```
1 root@debian_sa_minimal_linux:/boot/efi/EFI/debian$ mount |grep sda1
2 /dev/sda1 on /boot/efi type vfat (rw,relatime,fmask=0077,dmask=0077,
    ↪ codepage=437,iocharset=ascii,shortname=mixed,utf8,errors=remount-ro
    ↪ )
```

Nachdem nun auf die erste Festplatte auf die erste Partition gewechselt wurde, wird in der zweiten Zeile *grubx64.efi* ausgeführt. Die Ausführung von *file* in der Auflistung 2.5 zeigt, dass diese eine EFI-Applikation ist. In diesem Fall handelt es sich um den Grub-Bootloader, der den Kernel des Debiansystems ggf. mit bestimmten Parametern ausführt und die Kontrolle übergibt. Der Standard schreibt vor, dass alle Bootloader bei EFI im Verzeichnis *EFI/<vendor>/*.efi* abgelegt werden müssen[7].

Auflistung 2.5: file grubx64.efi

```
1 root@debian_sa_minimal_linux:/boot/efi/EFI/debian$ file grubx64.efi
2 grubx64.efi: PE32+ executable (EFI application) x86-64 (stripped to
    ↪ external PDB), for MS Windows
```

2.1.3 CD / DVD

Die optischen Medien verdienen eine separate Ausführung, da sie teilweise vom zuvor genannten abweichen. Wird eine CD / DVD für den BIOS- oder UEFI-Modus erstellt,

[7] Vgl. Pol11.

so muss diese dem *ISO 9660* oder dem *ISO 13346* Standard gerecht werden.
Der *ISO 9660* legt die Aufteilung eines optischen Mediums fest. Grob gesagt, kann die *ISO 9660* als Filesystem verstanden werden, wobei am Anfang ein System Bereich steht, der ggf. zum Ablegen der Bootinformationen für das BIOS genutzt wird. Dem schließt sich der Datenbereich an, der aus einem *Volume Discriptor Set* und dem darauf folgenden Pfad-Tabellen, Dateien und Verzeichnissen besteht[8]. Das Universal Disk Format (UDF), bzw. die *ISO 13346*, ist der Nachfolger von *ISO 9660* und räumt diverse Limitierungen von *ISO 9660* aus wie zum Beispiel die Partitionsgröße, die auf bis zu 2 TiB erweitert wurde, und die Länge von Dateinamen, die nun auf bis zu 254 Zeichen ausgeweitet wurde[9]. Der neue Standard fand seine Notwendigkeit in der wachsenden Kapazitäten und den steigenden Anforderungen an die optischen Medien.
BIOS nutzt den Systembereich des *ISO 9660*-Systems, um den Bootvorgang zu initiieren. Die Firmware, die UEFI unterstützt, ist in der Lage das Dateisystem der CD zu lesen und kann somit Standardrouten nutzen um den ggf. vorhandenen Bootloader zu starten.

2.1.4 Der Kernel

Die im folgenden dargestellten Codesegmente wurden dem Quellcode des Linux 4.9.130 Kernels entnommen.
Im Bootloader wurde automatisch durch einen Standardwert oder manuell ein Eintrag selektiert. Der Bootloader sucht nach dem komprimierten Kernel-Image und lädt dieses in den Speicher. Anschließend wird der Arbeitsspeicher für die Ausführung des Kernels vorbereitet, indem zum Beispiel die Commandline-Parameter für den Kernel und der lesbare Kernelversion-String im Arbeitsspeicher abgelegt werden. Ist die Arbeit des Bootloaders abgeschlossen, sind die vom Kernel-Header benötigten Paramter hinterlegt und das Programm springt zum Einstiegspunkt des *real-mode*-Kernels und führt die Methode *void main(void)* der */arch/x86/boot/main.c* aus. Es wird zum Beispiel die Erkennung des Speicher-Layouts und das Setzen des Videomodus durchgeführt, wie Auslistung 2.6 zeigt.

Auflistung 2.6: Auszug /arch/x86/boot/main.c

```
1 void main(void)
2 {
3  copy_boot_params();
4  ...
5  detect_memory();
6  ...
7  set_video();
8  go_to_protected_mode();
9 }
```

Bevor der Sprung zum *protected-mode* Kernel erfolgt, wird noch in der *go_to_protected_mode*-Funktion eine temporäre Interrupt Descriptor Table (IDT) und der Global Descriptor Table (GDT) des Speichers in jeweils ein bestimmtes CPU-Register geladen. Im *protected-mode* können nun bis zu 4 GB Speicherbei einer 32-bit-Architektur allokiert werden und

[8] Vgl. Bel95.
[9] Vgl. ubu17.

es erfolgt der Sprung auf den 32 Bit Einstiegspunkt, der Methode *startup_32* (siehe hierzu Auflistung 2.7).

Auflistung 2.7: Auszug /arch/x86/boot/pmjmp.c

```
1  ...
2  jmpl *%eax    # Jump to the 32-bit entrypoint
3  ENDPROC(in_pm32)
```

Die Funktion *startup_32* initialisiert die Register und löst anschließend durch den Aufruf der Methode *decompress* die Dekomprimierung des Kernels aus. Zudem gibt die Methode die Adresse des extrahierten Kernels zurück, zu der gesprungen wird. Der Status ändert sich von *"Decompessing ... done"* zu *"Booting the kernel"*. Auch hier wird zunächst eine *startup_32* Methode ausgeführt, die die letzten Vorbereitungen für den Kernel umsetzt und abschließend die *start_kernel*-Methode ausführt und damit zum letzten Teil des Bootvorgangs vom Kernel springt. Die Methode initialisiert eine Vielzahl von Subsystemen und Datenstrukturen, wie zum Beispiel den *Scheduler*, zu sehen in Auflistung 2.8.

Auflistung 2.8: Auszug /init/main.c: start_kernel()

```
1 asmlinkage __visible void __init start_kernel(void)
2 ...
3 mm_init();
4 /*
5 * Set up the scheduler prior starting any interrupts (such as the
6 * timer interrupt). Full topology setup happens at smp_init()
7 * time - but meanwhile we still have a functioning scheduler.
8 */
9 sched_init();
10 ...
```

Abschließend ruft sie die *rest_init*-Methode auf. *rest_init* erstellt einen neuen Kernel-Thread, übergibt diesem die Funktion *kernel_init* als Einstiegspunkt, startet den *Scheduler* um die Aufgabenverwaltung zu ermöglichen und geht abschließend in den Zustand *"idle"*, indem sie die Funktion *cpu_idle* aufruft. Im Anschluß übernimmt der *kernel_init* und initialisiert die restlichen CPUs. Zum Schluss versucht die Methode *init* oder zumindest die Bourne Shell zu starten, wie in Auflistung 2.9 zu sehen ist. Gelingt dies nicht, meldet der Kernel "Panic"[10]. Andernfalls ist das System nun in einem ggf. interaktiven Modus und wartet auf weitere Eingaben.

Auflistung 2.9: Auszug /init/main.c: kernel_init()

```
1 static int __ref kernel_init(void *unused)
2 {
3 ...
4  if (execute_command) {
5   ret = run_init_process(execute_command);
6    if (!ret)
7     return 0;
8    panic("Requested init %s failed (error %d).", execute_command, ret);
```

[10] Vgl. Dua08, : Wenngleich sämtliche Ausführungen aus dem Quellcode des Kernels entnommen werden können, bietet die Quelle eine hervorragende Zusammenfassung, auf die die Ausführungen im Paragraphen basieren.

```
 9  }
10  if (!try_to_run_init_process("/sbin/init") ||
11    !try_to_run_init_process("/etc/init") ||
12    !try_to_run_init_process("/bin/init") ||
13    !try_to_run_init_process("/bin/sh"))
14   return 0;
15  panic("No working init found. Try passing init= option to kernel. "
16     "See Linux Documentation/init.txt for guidance.");
17 }
```

Anmerkung zum Absatz: Die hier vorgestellte Mechanik bezieht sie auf eine 32-bit-Architektur und dient nur der exemplarischen Veranschaulichung des Prozesses. Eine 64-bit-Architektur würde beim Bootvorgang den größten Teil identisch durchlaufen. Zudem verwenden aktuelle Systeme zwar *systemd*, dennoch sucht der Kernel nach *init*, das in diesem Fall ein symbolischer Verweis auf das *systemd* darstellt wie in Auflistung 2.10 zusehen ist.

Auflistung 2.10: Ausgabe: file /sbin/init[11]

```
1  ~$ file /sbin/init
2  /sbin/init: symbolic link to /lib/systemd/systemd
```

Damit ist der Bootvorgang vom Einschalten bis zum Starten in eine interaktive Oberfläche abgeschlossen. Eben dieses Vorgehen gilt es im praktischen Teil umzusetzen. Es soll ein Medium entstehen, auf dem das Live-System wie oben beschrieben installiert ist und das in eine interaktive Oberfläche startet, aus der dann der Benchmark ausgeführt werden kann.

2.2 Die Anforderungen

Die Anforderungen an das Projekt ergaben sich zum einen durch den praktischen Einsatz des Systems und zum anderen wurden sie in den Besprechungen ausgearbeitet und präzisiert. Eine Auflistung aller Anforderungen sowie Abgrenzungen befindet sich im Anhang A.1. Hierbei wurde zwischen obligatorischen und optionalen Anforderungen unterschieden. Ebenfalls aufgeführt wurden die Abgrenzungen bzw. Funktionen, die im Resultat nicht notwendig sind und somit nicht weiter eingebracht wurden. Im Folgenden wird zunächst der praktische Einsatz des System beschrieben und es werden einzelne Anforderungen aufgeführt und genauer dargelegt.

2.2.1 Das Praktikum

Im Rahmen des Studiums der praktischen Informatik an der DHGE, müssen die Studierenden im 4. Semester das Praktium *Informationstechnik* Versuch *IT1*[12], *IT2*[13]

[11] Quelle: selbst erzeugte Ausgabe,
[12] Gri18a, Versuchsanleitung ist auf der CD hinterlegt.
[13] Gri18b, Versuchsanleitung ist auf der CD hinterlegt.

und *IT3*[14] durchführen.
Im ersten Versuch müssen die Studenten einen Rechner zusammenbauen. Dabei stehen ihnen verschiedene Bauteile wie die CPU, das Mainboard, der Arbeitsspeicher und das Netzteil zur Verfügung. Der Arbeitsspeicher soll in verschiedenen Konfigurationen wie zum Beispiel *Dual-* und *Singlechannelmode* montiert und getestet werden. Es geht darum ein Gefühl für eine sinnvolle Konfiguration zu bekommen und die Werte des Arbeitsspeichers, zum Beispiel Nenn- und Speichertakt, praktisch kennenzulernen und zu verstehen.
Der zweite Versuch beschäftigt sich mit Massenspeicher und RAID-Systemen. Dabei müssen die Studenten verschiedene Konfigurationen aufbauen und testen. So kommen diverse Schnittstellen wie SAS, SATA, USB2.0 und USB3.0 als auch Medien wie HDD und SSD zum Einsatz. Darüber hinaus werden RAID-Systeme unterschiedlicher Level wie RAID0, RAID1, RAID10 und RAID5 aufgebaut. Die Studenten prüfen dann die Übertragungsgeschwindigkeit beim Lesen und Schreiben der Konfigurationen. Anschließend werden Ausfälle simuliert und das RAID-System auf Konsistenz und Fehlertoleranz geprüft.
Im Versuch *IT3* wird die CPU und der Arbeitsspeicher übertaktet und wiederholt einem Benchmark unterzogen. Dabei gilt es herauszufinden, mit welchen Einstellungen das beste Ergebnis erzielt werden kann und bis zu welchem Punkt das System noch stabil betrieben werden kann.
Das Praktium beinhaltet zudem den Versuch *IT4*, der sich mit dem Thema Netzwerk beschäftigt und für diese Studienarbeit nicht relevant ist.

2.2.2 Obligatorische Anforderungen

Aus der Hardware des Praktikums ergeben sich bereits verbindliche Anforderungen. So soll das Zielsystem in der Lage sein mit den RAID-Controllern zu kommunizieren. Dafür werden die Treiber der Chips der Controller im Kernel eingebunden. Zudem muss der Kernel die RAID-Level verstehen. Darüber hinaus ist die Unterstützung die Schnittstellen USB2.0, USB3.0, SAS und SATA gefordert, sodass auch die Lese- und Schreibzugriffe auf die Massenspeicher ausgeführt werden können. Da in der Hochschule nur CPUs mit 64bit Architektur von Intel und AMD genutzt werden, entfällt die 32bit Kompatibilität. Zudem wird keine Crosscompilerchain benötigt, die verschiedene Befehlssätze der CPUs nutzbar macht. Hier wird der Fokus entsprechend der vorhandenen Hardware auf den x86-Befehlsatz gelegt. Um das System möglichst portabel zu halten, sollen sämtliche Linkvorgänge statisch erfolgen. Dies spielt gerade in der praktischen Umsetzung im Kapitel 3.2.2 eine bedeutende Rolle, wenn es um die Integration von *lsblk* geht. Das Modulloading beim Kernel soll vermieden werden. Der Bootvorgang sollte über UEFI erfolgen, kann aber auch alternativ über BIOS realisiert werden. Als optional gilt hierbei der Versuch sowohl UEFI als auch BIOS zu unterstützen.
Zu diesen spezifischen kommen weitere allgemeine Anforderungen hinzu. Da die Studenten mehrfach Tests der Hardware durchführen müssen nachdem sie im ausgeschalteten Zustand gewechselt wurde, benötigen sie ein System, dass zum einen schnell verfügbar ist, bzw. schnell bootet und zum anderen die eingesetzte Hardware möglichst wenig beansprucht und somit die Messungen aussagekräftiger werden. Die Ausgabe des Bench-

[14] Gri18c, Versuchsanleitung ist auf der CD hinterlegt.

marks soll direkt auf dem Monitor erfolgen, wodurch eine Möglichkeit zum Speichern der Ergebnisse entfällt. Da sich die Hardware der Testrechner in Zukunft wandeln könnte, besteht zudem der Bedarf nach einer flexiblen Lösung. Diese kann auch durch ein dokumentiertes Build-Skript realisiert werden. Um die Praktikabilität und fehlerfreie Arbeitsweise zu dokumentieren, soll gegen Ende der Studienarbeit ein Testlauf erfolgen, bei dem die RAID-Controller mit unterschiedlichen RAID-Level wie auch der CPU- und Arbeitsspeicher-Benchmark im Zielsystem getestet werden.

2.2.3 Optionale Anforderungen

Die Umsetzung der optionalen Anforderungen soll dynamisch zur Bearbeitung der Studienarbeit entschieden werden. Kernfaktoren sind dabei die Praktikabilität und der Nutzen.
Es kann ein *init*-System genutzt werden, bei dem wie bei einem normalen System der *init*-Prozess als Prozess 0 gestartet wird und sämtliche Aufgaben als Childs ausgeführt werden. Alternativ kann auch der Benchmark als Prozess 0 genutzt werden. Diesbezüglich könnten im Bootmenü verschiedene Einträge existieren, die je nach Wunsch anders starten. Die Auswahl würde dann folgende Punkte enthalten:

- Option 1: *init*-Start startet am Ende eine Shell in der Befehle, wie auch der Benchmark, ausgeführt werden können
- Option 2: Benchmark startet und wartet auf Eingaben
- Option 3: Benchmark wird durchgeführt und zeigt die Ergebnisse an

Wie bereits oben kurz erwähnt ist eine weitere optionale Anforderung die Unterstützung von BIOS und UEFI-Boot. Dadurch könnten Konfigurationsanpassungen im Bios des Testrechners entfallen, da beide Varianten unterstützt werden.
Eine weitere optionale Anforderungen geht der Frage nach, wie am leichtesten die Konfiguration des Kernels angepasst werden kann. Dabei wird hier die Idee vertreten, dass vorgefertigte *.config*-Dateien existieren, die jeweils immer nur ein Feature integrieren. Die Dateien wurden als *diff*-Datei gespeichert. Der Befehl *diff*[15] vergleicht zeilenweise zwei Dateien miteinander und kann das Resultat als *diff*-Datei speichern, sodass durch den Befehl *patch*[16] eine spätere Zusammenführung wieder stattfinden kann.

2.2.4 Abgrenzungen und unnötige Funktionen

Der Umriss des Praktikums und die obligatorischen und optionalen Anforderungen zeichnen bereits ein recht genaues Bild. Dadurch ergeben sich Punkte, die im späteren System nicht notwendig oder gar nicht erwünscht sind.
Dadurch, dass sämtliche Treiber und Programme im Image bereits integriert sind, entfällt jede Notwendigkeit für Netzwerkunterstützung. Ferner könnte dieses Feature eher Fehler produzieren, da das speziell zugeschnittene System sich für die Anwender

[15] http://man7.org/linux/man-pages/man1/diff.1.html
[16] http://man7.org/linux/man-pages/man1/patch.1.html

sicherlich ungewohnt verhalten wird. Da das System schnell starten und die vorhandene Hardware wenig beanspruchen soll, kommt eine graphische Oberfläche nicht in Frage. Zudem wird die Eindeutigkeit der Nutzbarkeit durch die Console erhöht und es kommt zu keinen falschen "Klicks". Der Bootvorgang muss kein *Secure UEFI* beherrschen. Es reicht ein unsignierter Bootloader.

Die hier vorgestellten Anforderungen bieten einen Rahmen in dem das Projekt realisiert werden kann. Dabei können auch während der Ausarbeitung weitere Funktionen hinzugefügt werden, solange diese nicht einem der hier aufgeführten Punkte widersprechen. Sollte dies dennoch der Fall sein, wird im praktischen Teil eine Begründung zu der Entscheidung aufgeführt.

2.3 Das Werkzeug

Im Internet gibt es bereits weit etablierte Systeme, die dem Vorhaben der Projektarbeit zuträglich sind und damit einen Großteil der Arbeit abnehmen, sodass sich bei der praktischen Umsetzung auf die Erweiterung bzw. auf die Anpassung des Werkzeugs konzentriert werden kann. Bei dem vorliegenden Projekt wurde *Minimal-Linux-Live*[17] verwendet. Das System verfügt über eine ausführliche Dokumentation und lässt durch die Verwendung eines Shell-Skripts zum Erstellen des Images das Nachvollziehen der einzelnen Befehle und des gesamt Prozesses zu. Da die ausführliche Handhabung, der Ablauf und die Anpassung im Kapitel 3 ausgeführt werden, wird hier nur eine kurze Erwähnung vorgenommen.

2.3.1 Die Alternative

Bevor die Entscheidung auf *Minimal-Linux-Live* fiel, wurden zunächst eine manuelle Durchführung und anschließend *Ycoto* ausprobiert. Da sich bereits schnell zeigte, dass *Minimal-Linux-Live* die angemessene Variante ist, wurden keine ausführlichen Testläufe mit den anderen Systeme unternommen, so dass eine detaillierte Gegenüberstellung an dieser Stelle nicht möglich ist.

Die manuelle Durchführung Bei der manuellen Durchführung wurde zunächst eine virtuelle Maschine mit Debian 4.9.0-8 Kernel aufgesetzt. In dieser wurde über kernel.org der neuste Kernel-Quellcode heruntergeladen und extrahiert. Um den Kernel bearbeiten und kompilieren zu können, mussten folgende Pakete installiert werden:

- ncurses-devel, qt-devel or unifdef
- libelf-dev, libelf-devel or elfutils-libelf-devel
- bc
- flex

[17] http://minimal.linux-bg.org/

- bison
- libssl-dev

Anschließend konnte durch den Aufruf von *make nconfig* - alternativ auch *make menuconfig* oder *make xconfig* - im Ordner des extrahierten Kernels das Konfigurationsmenü für den Kernel gestartet werden (siehe hierzu Abb. 2.1). In diesem wurden nach einan-

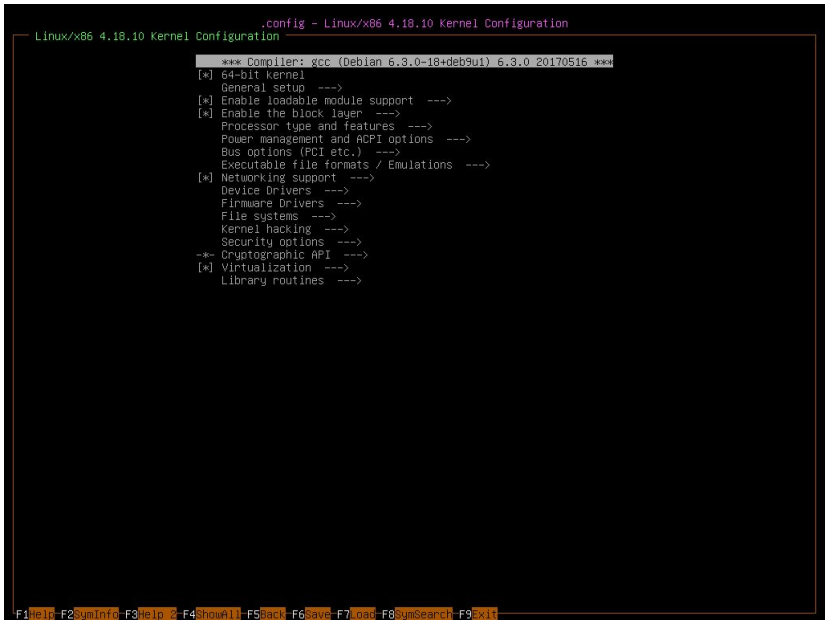

Abbildung 2.1: make nconfig bei der manuellen Durchführung[18]

der alle Punkte durchgegangen und entsprechend der Anforderungen hinzugefügt oder abgewählt. Dabei ist anzumerken, dass die Mehrzahl der Einträge eine Hilfe offeriert, die bei der Entscheidung hilfreich war. Abschließend wurde die Konfiguration gespeichert. Durch den Befehl *make install* wurde der Kernel mit der neuen Konfiguration kompiliert und unter */boot/* als neuer komprimierter Kernel gespeichert. Zudem wurde ein Eintrag in der EFI-Firmware-Konfiguration für den Kernel hinterlegt wie in Auflistung 2.11 zu sehen ist.

Auflistung 2.11: Testmaschine: make install[19]

```
1 root@debian_sa_custom_made:/home/test/linux-4.18.19# make install
2 sh ./arch/x86/boot/install.sh 4.18.10 arch/x86/boot/bzImage \
3 System.map "/boot"
4 run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 4.18.10 /
      ↪ boot/vmlinuz-4.18.10
5 run-parts: executing /etc/kernel/postinst.d/initramfs-tools 4.18.10 /boot
      ↪ /vmlinuz-4.18.10
6 update-initramfs: Generating /boot/initrd.img-4.18.10
7 ...
8 run-parts: executing /etc/kernel/postinst.d/zz-update-grub 4.18.10 /boot/
      ↪ vmlinuz-4.18.10
9 Generating grub configuration file ...
10 ...
11 Found linux image: /boot/vmlinuz-4.9.0-8-amd64
12 Found initrd image: /boot/initrd.img-4.9.0-8-amd64
```

[18] Quelle: selbst erstellte Grafik

```
13 Adding boot menu entry for EFI firmware configuration
14 done
```

Anschließend wurde die virtuelle Maschine neu gestartet und mit dem neuen Kernel ausgeführt. Dabei kam es jedoch zu einer fehlerhaften visuellen Ausgabe. Erst als der Kernel mit der Standardkonfiguration - mittels *make defconfig* erstellt - kompiliert wurde, konnte ein weitestgehend fehlerfreier Durchlauf bewerkstelligt werden.
Durch dieses Vorgehen konnte eine Konfiguration erstellt und umgehend getestet werden, ohne dass dazu ein Image erzeugt und eingebunden werden musste. Darum eignete sich die Methode, um Anpassungen an der Konfiguration vom Kernel zu testen und wurde daher auch im weiteren Verlauf der Studienarbeit genutzt. Zudem offerierte das Durcharbeiten jedes einzelnen Punktes der Konfiguration einen sehr weitreichenden und doch tiefen Einblick in den Kernel und war daher ebenso von Vorteil für den weiteren Verlauf. Nachdem nun ein funktionsfähiger Kernel erstellt werden konnte, stellte sich die Frage, wie ein ISO-Image gebaut werden kann. Die Recherche brachte *Minimal-Linux-Live* hervor, das die notwendigen Kommandos inklusive der Dokumentation bot um die Arbeit fortzusetzen. Dennoch wurden Alternativen in Betracht gezogen und angetestet.

Vergleichswerte Zum Zeitpunkt, als weitere Methoden in Betracht gezogen wurden, konnte bereits ein Image und ein Kernel mit unterschiedlichen Konfiguration erstellt werden. Das Image wurde in einer virtuellen Maschine getestet. Dadurch ließen sich Werte wie die Bootzeit[20] und die Größe des Images wie auch der Kernels festhalten. Diese Werte und die generelle Handhabung der Methode wie auch die Abdeckung der Anforderungen wurde als Vergleichswerte benutzt.

Yocto *Yocto*[21] ist ein Projekt mit dem, ähnlich wie bei *Minimal-Linux-Live*, ein hochspezialisiertes Live-System erstellt werden kann. Es ist benutzerfreundlich dokumentiert und bietet beachtliche viele Anpassungsmöglichkeiten. Beim Testlauf wurde eine neue virtuelle Maschine mit Debian 4.9.0-8 aufgesetzt. Bevor das Bauen des Images umgesetzt werden kann, setzt *Yocto* folgende Versionen voraus:

- Git >= 1.8.3.1
- tar >= 1.27
- Python >= 3.4.0

Zudem müssen die in Auflistung 2.12 aufgelisteten Pakete installiert werden.

Auflistung 2.12: Yocto: benötigte Pakete[22]

[19] Quelle: selbst erstellte Ausgabe
[20] Bootzeit: ist die Zeit, die vom Einschalten bis zum interaktiven Modus vergeht, sozusagen die Geschwindigkeit des Startvorgangs.
[21] https://www.yoctoproject.org/
[22] Quelle: Yoc18.

```
1 $ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-
    ↪ multilib build-essential chrpath socat cpio python python3 python3-
    ↪ pip python3-pexpect xz-utils debianutils iputils-ping libsdl1.2-dev
    ↪  xterm
```

Sind die Vorbereitungen abgeschlossen, kann über das Git-Repository *Poky*[23] der Quellcode bezogen werden. In dem Repository befindet sich die Datei *oe-init-build-env*, die die Umgebungsvariablen für das Build-Skript anlegt und anschließend den User wie in Auflistung 2.13 dargestellt, über das weitere Vorgehen informiert.

Auflistung 2.13: Yocto: oe-init-build-env[24]

```
1 test@debian_sa_yocto:~/poky$ source oe-init-build-env
2 ### Shell environment set up for builds. ###
3 You can now run 'bitbake <target>'
4 Common targets are:
5 core-image-minimal
6 core-image-sato
7 meta-toolchain
8 meta-ide-support
9 You can also run generated qemu images with a command like 'runqemu
    ↪ qemux86'
```

Abschließend kann durch den Befehl *bitkake core-image-minimal* ein Image erzeugt werden.
Das Image wurde in verschiedenen Konfigurationen erstellt und in der virtuellen Maschine getestet. Die Werte für die Geschwindigkeit beim Starten und die Größe des Images blieben selbst mit dem Target *core-image-minimal* hinter den des Images von *Minimal-Linux-Live* zurück. Zudem benötigte das Image zum Bauen wesentlich mehr Zeit als die Referenz. Darüber hinaus wurden bei beiden Methoden die Anforderungen abgedeckt und die Benutzung fiel ebenso gleich aus. Bei der Arbeit mit *Yocto* hatte man stets den Eindruck eines zu potenten Werkzeugs für die vorliegenden Aufgabe. Daher wurde es für die weitere Bearbeitung nicht genutzt.

Weitere Systeme wurde innerhalb der Arbeit nicht evaluiert oder getestet. Die Recherche wurde zwar noch fortgesetzt, jedoch kamen dabei keine nennenswerten Funde bei heraus. Zudem sei an dieser Stelle angemerkt, dass die gewählte Methode eine überaus komfortable und leicht zu erweiternde Basis bietet, die darüber hinaus gut dokumentiert ist und somit eine tiefere Suche nach anderen Methoden schwer zu begründen ist.

[23] https://git.yoctoproject.org/git/poky
[24] Quelle: selbst erzeugte Ausgabe

3 Die Realisierung

Nachdem nun im zweiten Kapitel der generelle Startprozess und die Anforderungen dargelegt wurden, findet in diesem Kapitel die Dokumentation der Realisierung statt. Dafür wird zunächst *Minimal-Linux-Live* beschrieben und dessen Vorgehensweise ausführlich dargelegt. Anschließend werden die Anpassungen an der Kernelkonfiguration und am Build-Skript ausgeführt. Zuletzt wird der Testlauf im Labor beschrieben.

3.1 Minimal-Linux-Live

Minimal-Linux-Live[1] ist ein Projekt von Ivan Davidov, der sich ein eigenes Linux OS erstellen wollte und dabei die Essenz seiner Recherche in dieses Projekt formte[2]. Es bietet die Möglichkeit durch ein Shell-Script ein voll funktionsfähiges Livesystem zu erstellen, das auf einen USB-Stick kopiert oder eine CD gebrannt werden kann. Anschließend kann von dem Medium in eine interaktive Oberfläche gebootet werden. Im Folgenden wird die Vorgehensweise des Build-Skriptes beschrieben.

3.1.1 Die Voraussetzungen

Bei der Suche nach dem Quellcode von MLL stößt man auf zwei Varianten. Die erste Variante [3] enthält dabei ein Skript über das das gesamte Build durchgeführt wird. Die zweite Variante[4] liefert zwar auch ein Hauptskript, dennoch sind alle Aufgaben auf separate Skripte aufgeteilt und lassen sich somit problemlos integrieren. Darüber hinaus ermöglicht es die Integration von eigenen Einstellungen und Overlays. Um die grundlegende Vorgehensweise beim Build zu erklären, ist die Variante 1 ausreichend.
Um das Skript reibungslos ausführen zu können, sollte via *git clone* das Git-Repository von Variante 1 lokal gespeichert werden. Darüber hinaus benötigt das Skript und insbesondere der erweiterte Skript aus dem zweiten Teil, folgende Pakete, die mittels *sudo apt install* installiert werden können:

- install
- wget
- bc

[1] im Folgenden mit *MLL* abgekürzt
[2] Vgl. Dav18.
[3] https://github.com/ivandavidov/minimal-linux-script
[4] https://github.com/ivandavidov/minimal

- build-essential
- gawk
- xorriso
- bison
- flex
- libelf-dev
- libssl-dev
- ncurses-devel oder libncurses5-dev

Anschließend kann durch den Befehlt *./minimal.sh* der Build Prozess gestartet werden.

3.1.2 Das Build-Skript

Das Build-Skript *minimal.sh* kann in drei Phasen eingeteilt werden: die Vorbereitung, die Konfiguration und das Bauen der Komponenten und die Zusammenführung. Zudem ist es für die *Bourne Shell* oder *Dash* geschrieben und sollte dadurch auf den meisten Linux-Distributionen fehlerfrei ausführbar sein.
Sämtliche Auflistungen von Shell-Skriptinhalten in diesem Teil des Kapitels, soweit nicht anders ausgewiesen, wurden dem Shell-Skript[5] von Ivan Davidov entnommen.

Die Vorbereitung In der Vorbereitung werden lokale Variablen gesetzt und die benötigten Pakete herunterladen und extrahiert. Im Wesentlichen besteht das Ergebnis aus einem *Linux-Kernel*, einer *Busybox* und einem *Syslinux*.
Um die einzelnen Versionen der Module schnell anzupassen, werden die Versionsnummern ganz zu Beginn festgelegt, wie in Auflistung 3.1 zu sehen ist.

Auflistung 3.1: Basisskript: Versionsfestlegung

```
1 KERNEL_VERSION=4.12.3
2 BUSYBOX_VERSION=1.27.1
3 SYSLINUX_VERSION=6.03
4 wget -O kernel.tar.xz http://kernel.org/pub/linux/kernel/v4.x/linux-
    ↪ $KERNEL_VERSION.tar.xz
5 wget -O busybox.tar.bz2 http://busybox.net/downloads/busybox-
    ↪ $BUSYBOX_VERSION.tar.bz2
6 wget -O syslinux.tar.xz http://kernel.org/pub/linux/utils/boot/syslinux/
    ↪ syslinux-$SYSLINUX_VERSION.tar.xz
7 tar -xvf kernel.tar.xz
8 tar -xvf busybox.tar.bz2
9 tar -xvf syslinux.tar.xz
10 mkdir isoimage
11 ...
```

Als nächstes werden die einzelnen komprimierten Tarballs der Module heruntergeladen und extrahiert, so dass diese nun im Arbeitsordner[6] existieren und später genutzt

[5] Dav18.
[6] Arbeitsordner: Bezeichnung für den Ordner in dem ein Skript ausgeführt wird und der somit als Bezugspunkt für alle weiteren Aktionen dient

werden. Zum Abschluss der Vorbereitung wird noch das Verzeichnis *isoimage* erstellt, in dem zu einem späteren Zeitpunkt die konfiguriert und erstellten Module kopiert werden. Auflistung 3.2 zeigt den Inhalt des Arbeitsordners nach diesen Aktionen.

Auflistung 3.2: Basisskript: Ordnerstruktur nach Vorbereitung[7]

```
1 test@debian_sa_final_test:~/minimal-linux-script$ ls
2 busybox-1.27.1 busybox.tar.bz2 clean.sh isoimage kernel.tar.xz LICENSE
    ↪ linux-4.12.3 minimal.sh qemu64.sh README.md syslinux-6.03 syslinux.
    ↪ tar.xz
```

Die Konfiguration und der Bau Als erstes wird in den zuvor extrahierten Busybox-Ordner gewechselt und die *Busybox* konfiguriert und kompiliert. Dabei wird durch den *make distclean defconfig* eine Standardstruktur angelegt und eine Standardkonfiguration für die Kompilierung erstellt[8]. Damit die Busybox ohne weitere Bibliotheken verwendet werden kann, wird der Wert *CONFIG_STATIC* der Konfigurationsdatei *.config* auf *y* abgeändert und sorgt dadurch für eine statische Linkung beim Kompilieren. Abschließend wird das Kommando *make busybox install* ausgeführt, wobei der Parameter *busybox* für das Erstellen der Binärdatei sorgt. Durch den Parameter *install* werden symbolische Links für die in der Konfiguration freigeschalteten Kommandos erstellt, die alle auf die Binärdatei *busybox* zeigen. So wird zusnächst in das *_install*-Verzeichnis gewechselt. Die Auflistung 3.3 zeigt die bis hierhin vorgestellten Schritte im Skript.

Auflistung 3.3: Basisskript: Bau und Konfiguration Busybox

```
1 ...
2 make distclean defconfig
3 sed -i "s/.*CONFIG_STATIC.*/CONFIG_STATIC=y/" .config
4 make busybox install
5 ...
```

Nachfolgend werden nun Änderungen an dem Standardbuild vorgenommen. Durch die aufgerufene Make-Funktion wurde ein Verzeichnis *_install* angelegt, in dem die Basisstruktur für die *Busybox* gebaut wurde. Als erstes wird die Datei *linuxrc* entfernt. Diese sorgt für einen Aufruf von *discover* und sorgt damit für die Hardwareerkennung und -installation. Danach werden die Standardverzeichnisse für das *Root-Filesystem* angelegt und ein ausführbares *init*-Shell-Skript erstellt. Die Datei mountet die Systemordner als Pseudofilesystem auf die eben zuvor erstellten Ordner und startet danach in eine Shell. Das Skript macht die *init*-Datei ausführbar und fügt dann sämtlichen Inhalt von *_install* in einem *cpio*-Archiv zusammen. Zuletzt wird die Ausgabe komprimiert im zuvor erstellten *isoimage*-Ordner als *rootfs.gz* abgespeichert, wie die Auflistung 3.4 zeigt. Dadurch ist das erste Modul abgeschlossen und das System verfügt über ein Root-Filesystem.

Auflistung 3.4: Basisskript: Erstellung des Rootfs

```
1 ...
```

[7] Quelle: selbst erzeugte Ausgabe
[8] Quelle: Aus der Ausgabe von *make help* im Busysbox-Ordner entnommen

```
2 cd _install
3 rm -f linuxrc
4 mkdir dev proc sys
5 echo '#!/bin/sh' > init
6 echo 'dmesg -n 1' >> init
7 echo 'mount -t devtmpfs none /dev' >> init
8 echo 'mount -t proc none /proc' >> init
9 echo 'mount -t sysfs none /sys' >> init
10 echo 'setsid cttyhack /bin/sh' >> init
11 chmod +x init
12 find . | cpio -R root:root -H newc -o | gzip > ../../isoimage/rootfs.gz
13 ...
```

Als nächstes wird der Kernel gebaut. Dabei löscht der Parameter *mrproper* alle vorhanden Konfigurationdateien und alle generierten Dateien. Erneut wird durch den Parameter *defconfig* eine Standardkonfiguration erstellt. Der letzte Parameter sorgt letztendlich dafür, dass das Kernel gebaut und in ein *bzImage* gepackt wird, das im nächsten Schritt als *kernel.gz* der Isoimage-Zusammenstellung hinzugefügt wird (siehe hierzu Auflistung 3.5)[9]. Somit wurde auch das zweite Module der Kollektion hinzugefügt.

Auflistung 3.5: Basisskript: Bau des Kernels

```
1 ...
2 cd ../../linux-$KERNEL_VERSION
3 make mrproper defconfig bzImage
4 cp arch/x86/boot/bzImage ../isoimage/kernel.gz
5 ...
```

Es gibt nun den Kernel, der angesprochen werden kann und eine Root-Filesystem, dass durch den Kernel verwendet werden kann. Es fehlt noch das Modul, das den Boot über das Medium ermöglicht. Hier tritt *syslinux* auf den Plan. Es wird der Bootloader zur Kollektion hinzugefügt und eine Konfigurationsdatei geschrieben, in der auf den Kernel und das Root-Filesystem verwiesen wird (siehe hierzu die Auflistung 3.6).

Auflistung 3.6: Basisskript: Integration Syslinux

```
1 ...
2 cd ../isoimage
3 cp ../syslinux-$SYSLINUX_VERSION/bios/core/isolinux.bin .
4 cp ../syslinux-$SYSLINUX_VERSION/bios/com32/elflink/ldlinux/ldlinux.c32 .
5 echo 'default kernel.gz initrd=rootfs.gz' > ./isolinux.cfg
6 ...
```

Die Zusammenführung Damit ist auch das letzte Modul Teil der Kollektion. Am Ende des Skriptes sorgt *xorriso*, wie in Auflistung 3.7 abgebildet, für den letztendlichen Bau der ISO-Datei. Der erste Parameter gibt an, dass das Iso-Image mit speziellen Eigenschaften gebaut werden soll, die nun weiter aufgeführt werden[10]: *-o* gibt den Ausgabepfad und -namen an; *-b* spezifiziert das Bootimage, das im Bootkatalog hinterlegt werden soll

[9] Quelle: Der Ausfgabe von *make help* im Kernel-Ordner entnommen.
[10] Vgl. Sch16a.

und als bootbar markiert wird; *-c* gibt die Adresse des Bootkatalogs im Image an; *-no-emul-boot* gibt an, dass das Image im aktuellen Katalog keine Diskette oder Festplatte emuliert; *-boot-load-size* gibt die Anzahl an 512Bytes-Blöcken an, die beim Booten vom Bootimage geladen werden sollen; *-boot-info-table* überschreibt die Bytes 8 bis 63 des aktuellen Bootimages; der *./* am Ende sorgt dafür, dass die Dateien und Ordner des aktuellen Verzeichnisses im Image verbaut werden.[11] Durch diesen letzten Befehl wird das ISO-Image erstellt und kann nun weiterverwendet werden.

Auflistung 3.7: Basisskript: ISO-Imagebau durch Xorriso

```
1 ...
2 xorriso \
3 -as mkisofs \
4 -o ../minimal_linux_live.iso \
5 -b isolinux.bin \
6 -c boot.cat \
7 -no-emul-boot \
8 -boot-load-size 4 \
9 -boot-info-table \
10 ./
11 ...
```

3.1.3 Das Ergebnis

Nachdem das Skript durchgelaufen ist, befindet sich im Ordner ein ISO-Image namens *minimal_linux_live.iso* (siehe hierzu Auflistung 3.8), das auf ein optisches Medium gebrannt, auf einem USB-Stick übertragen oder in einer virtuellen Maschine als CD eingebunden werden kann.

Auflistung 3.8: Basisskript: ISO-Datei[12]

```
1 test@debian_sa_final_test:~/minimal-linux-script$ file minimal_linux_live
    ↪ .iso
2 minimal_linux_live.iso: ISO 9660 CD-ROM filesystem data 'ISOIMAGE' (
    ↪ bootable)
```

Dabei ist zu beachten, dass dieses Skript ein Image produziert, dass über BIOS-Boot aufgerufen werden muss. Eine UEFI-Umsetzung ist mit diesem Skript ohne Anpassungen nicht möglich.
Wird das Image als Medium in einer virtuellen Maschine eingelegt und die Maschine führt Bios-Boot durch, so kann nach bereits 6 Sekunden auf eine interaktive Shell zugegriffen werden. Die Abbildung reffig:basisFirstBoot zeigt den Startbildschirm nachdem der Bootvorgang abgeschlossen ist.

[11] Vgl. Sch16b.
[12] Quelle: selbst erzeugte Ausgabe

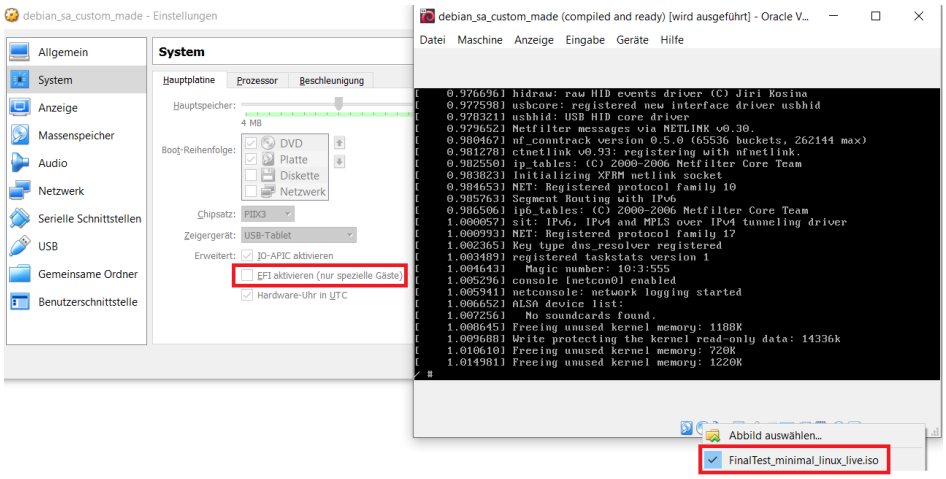

Abbildung 3.1: Basisskript: Start der VM mit erzeugtem Image[13]

3.2 Die Anpassungen

Nachdem nun der grundsätzliche Bau von *MLL* ausführlich dargestellt wurde, werden im Folgenden die Anpassung beschrieben, die aus den Anforderungen resultierten. Dafür wurde ein separates Skript geschrieben, dass auf der Basis von *MLL* aufbaut. Daher sind die folgenden Skriptzeilen, soweit nicht anders angeführt, aus diesem entnommen.

3.2.1 Die Kernelkonfiguration

Im Kapitel 2.2.2 wurden die obligatorischen Anforderungen erklärt und beschrieben, die nun Teil der Kernelkonfiguration werden sollen.
Die Ausführung des Befehls *make defconfig* erzeugt im Kernelverzeichnis eine Standardkonfiguration, die als Ausgangsbasis für die weiteren Anpassungen dient. Diese wird in der Datei *.config* gespeichert und kann durch den Befehl *make config* oder alternativ durch *make menuconfig* oder *make nconfig* abgeändert werden. Die Abbildung 3.2 zeigt, dass der erste Befehl rein textbasiert ist, der zweite ein rudimentäres Auswahlmenü bietet und der dritte eine weitere Möglichkeit eines Auswahlmenüs anbietet.

In dieser werden nun die Einstellungen eingepflegt.

Obligatorische Anforderungen Um die RAID- und SAS-Kontroller und die Schnittstellen zu unterstützen, müssen die dazugehörigen Treiber im Kernel integriert werden. Gleiches gilt für die Unterstützung der textitRAID-Level. Dabei befinden sich die einzelnen Treiber unter folgenden Unterpunkten:

- SAS: wird automatisch hinzugefügt

[13] Quelle: selbst erstellte Grafik

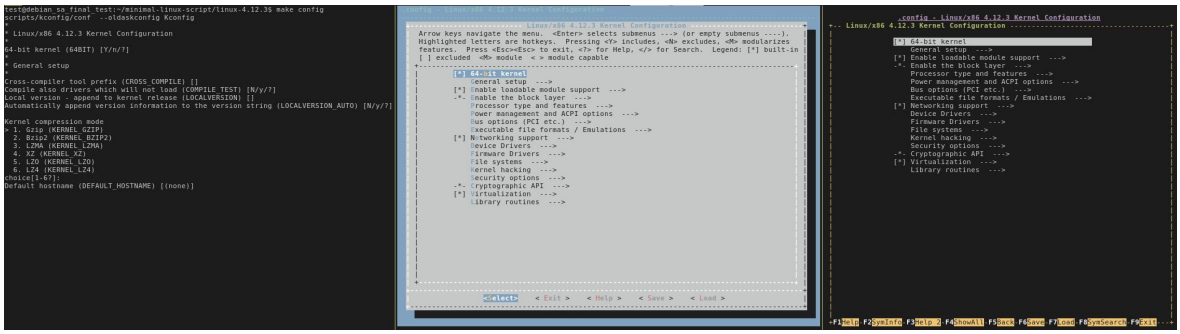

Abbildung 3.2: Bearbeiten der Kernelkonfiguration: make config (links), make menuconfig (mitte) und make nconfig (rechts)[14]

- SATA: Device Drivers => Serial ATA and Parallel ATA drivers (libata) => AHCI SATA Support
- USB: => Device Drivers => USB Support
  - V1.0: OHCI HCD (USB 1.1) support
  - V2.0: EHCI HCD (USB 2.0) support
  - V3.0: xHCI HCD (USB 3.0) support
- SAS-Controller:
  - LSISAS1068E:
    => Device Drivers => Fusion MPT => Fusion MPT ScsiHost Drivers for SAS
- RAID-Controller
  - Darwin Control DC-324e RAID: Maevell 88SX7042
    => Device Drivers => Serial ATA and Parallel ATA drivers (libata) => Marvel SATA Support
  - LSI 3ware SAS 9750-4i: LSISAS2108 RAID-on-Chip (ROC)
    => Device Drivers => SCSI device support => SCSI low-level device => 3ware 97xx
  - Adaptec ASR-6805: Microsemi PM8013 Dual Core RAID-on-Chip
    Dieser Treiber wurde leider nicht gefunden. Auch ein Ersatztreiber unterstützte die Karte nicht, wie der Testlauf später bestätigte.
- RAID-Level
  => Device Drivers => Multiple devices driver support (RAID and LVM)
  Darunter befinden sich die einzelnen *RAID*-Level

Alle gewünschten Treiber werden durch einen Stern markiert und somit nicht als Modul sondern direkt in den Kernel geladen.
Die 64bit Architektur kann direkt im obersten Menü ausgewählt werden. Für die CPU-Unterstützung wird der *Generic x86-64*-Treiber genutzt, der unter *=> Processor type and features => Processor family (Generic-x86-64)* ausgewählt werden kann. Alternativ stehen hier auch separat die Intel- und der AMD-CPU zur Verfügung. Da aber nur einer von beiden geladen werden kann, fiel die Entscheidung auf den generischen Treiber.

[14] Quelle: selbst erstellte Grafik

Optionale Anforderungen Die im Dokument festgehaltenen optionalen Anforderungen werden außerhalb der Konfiguration des Kernels realisiert. Dennoch sei an dieser Stelle angemerkt, dass unter *=> Device Drivers => Graphics Support* diverse Grafikkartentreiber eingebunden werden können. Durch die Standardkonfiguration unterstützt der Kernel zumindest die Ausgabe via VGA, wie im Testlauf nachgewiesen wurde.

Anmerkung Durch die Standardkonfiguration kann mit wenigen Schritten ein funktionsfähiges System geschaffen werden. Dennoch fehlen die oben aufgeführten Treiber, die für das Praktikum notwendig sind. Darüber hinaus wurden diverse Einstellungen, wie zum Beispiel die Netzwerkunterstützung unter *=> Device Driver => Network Device Support* entfernt, um den Kernel kleiner und schneller zu machen. Dennoch sollte bei der Anpassung stets mit Bedacht vorgegangen werden. Es ist in vielen Punkten schwierig die Tragweite der Deaktivierung zu erkennen, sodass im Zweifelsfall der Treiber oder das Feature lieber aktiviert bleiben sollte. Um einen Ansatz für eine Recherche zu erhalten, kann bei vielen Optionen eine Hilfe aufgerufen werden, die Details zur Implementierung und Nutzung preisgibt.

3.2.2 Das Messwerkzeug

Im Laufe der Studienarbeit konnte das Messwerkzeug an diesen Teil der Arbeit für die Testläufe überreicht werden. Dabei handelt es sich um den Quellcode von Falko Linke, der mittels *Make*-Datei die benötigten Binärdateien statisch kompiliert. Das Kompilat besteht aus vier Dateien, die im späteren System im Verzeichnis *bin* abgelegt werden müssen.

- hardwareBenchmark: Das Menü zur Auswahl des Tests
- ramTest: Der Arbeitsspeichertest
- blockDeviceTest: Der Test für die Medien wie HDD, SSD und RAID
- cpuTest: Der CPU-Test

Eine Anforderung des Entwicklers war, dass das Programm *lsblk* unterstützt werden müsse. *lsblk* gibt diverse Informationen zu Blockgeräten aus[15]. Die Ansicht sollte dabei, wie in Auflistung 3.9 dargestellt, ausgeben werden. Im späteren Test konnte dann der User anhand der Übersicht entscheiden, welches Gerät im *blockDeviceTest* getestet werden sollte.

Auflistung 3.9: Ausgabe lsblk[16]

```
1 hardwaretestbenchmark-dhge$ lsblk -ld -o +VENDOR,MODEL,TRAN
2 NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT VENDOR   MODEL       TRAN
3 sda    8:0    0   20G 0 disk             ATA    VBOX HARDDISK sata
4 sr0   11:0    1 1024M 0 rom                VBOX   CD-ROM      ata
```

[15] für näher Informationen siehe Manpage zu lsblk
[16] Quelle: Selbst erzeugte Ausgabe

lsblk ist ein Modul aus dem *util-linux*-Paket. Um dieses kompilieren zu können, muss der Quellcode[17] heruntergeladen werden. Anschließend wird durch das Kommando *./configure* die Grundstruktur und Konfigurationsdateien erstellt. Durch den Befehl *make lsblk* wird das Kompilat für *lsblk* erstellt. Dieses wurde in das Image eingebunden, jedoch konnte es in der gestarteten Liveumgebung nicht ausgeführt werden. Dies liegt darin begründet, dass die erzeugte Datei nur ein *Wrapper* um die Binärdatei herum ist. Aber selbst mit der Binärdatei wäre die Ausführung auf dem Live-System nicht ohne weiteres möglich, da diese dynamisch gelinkt wurde. Dadurch kollidiert die Anforderung des Entwicklers mit den Anforderungen an das System. Als Lösung wurde mittels *strace* das Verhalten von *lsblk* untersucht. Dabei konnten die Dateien ausfindig gemacht werden, die *lsblk* nutzte um die Information der Medien zu erhalten. Diese konnten im hardwareBenchmarktest integriert werden und sorgten somit für eine angemessene Ausgabe ohne dynamisch gelinkte Binärdateien.

3.2.3 Das Shell-Skript

Das Basis Shell-Skript, dass von Ivan zur Verfügung gestellt wird und weiter oben näher beschrieben wurde, ist statisch organisiert. Wenn Änderungen, wie zum Beispiel eine andere Kernelversion, gemacht werden, müssen diese stets im Skript an der korrekten Stelle erfolgen. Damit diese Statik aufbricht, wurde das Skript mit Optionen und Parametern ergänzt. Dabei ist der Grundgedanke folgender: Werden keine Parameter übergeben, wird die Standardkonfiguration gebaut und ausgeliefert. Bis auf weniger Änderungen gleicht diese Ausgabe der des Basisskriptes. Wird ein Parameter übergeben, ersetzt der Wert den Standardwert und wird entsprechend gehandhabt. Dabei besteht natürliche keine Garantie für die Funktionsfähigkeit des erzeugten Ergebnisses. Im Folgenden soll ein Teil der Anpassungen aufgeführt werden, um ein Verständnis von der erweiterten Handhabung des Skriptes zu bekommen.

Die Optionen Es zeigte sich, dass der Kernel nicht bei jedem Durchlauf neu heruntergeladen werden muss. Daher wurde der Parameter *-K* eingeführt. Diesem kann entweder ein Tarball zum entpacken oder direkt der entpackte Kernel-Ordner übergeben werden. Damit wird der Prozess des Herunterladens übersprungen und mit einer Kopie des vorhandenen Kernels gearbeitet. Zudem wurde die Möglichkeit offeriert durch die Option *-c* einen Konfigurationsmodus beim Kompilieren des Kernels zu nutzen. Dabei kann optional eine Kernelkonfigurationsdatei angegeben werden. Ist dies der Fall muss der Modus entweder *oldnoconfig* oder *oldyesconfig* sein. Bei ersteren werden alle nicht gesetzten Optionen mit *no* beantwortet, bei letzteren mit *yes*. Diese Option gibt die Möglichkeit einen Kernel auf Basis einer bereits erprobten Konfiguration zu erstellen, wodurch die Kompatibilität erhöht und die Fehlerquote des neuen Kompilats reduziert wird. Sollte sich der Benutzer dennoch für einen Download entscheiden, kann er über die Option *-k* eine Kernelversionsnummer angeben, die bezogen wird.
Abseits dessen kann der Nutzer den Workspace über die Option *-w* bestimmen. Diese Option kommt gerade dann zu tragen, wenn auf der bestehenden Partition nicht mehr genügend Kapazität vorhanden ist und somit auf eine andere Partition ausgewichen

[17] https://mirrors.edge.kernel.org/pub/linux/utils/util-linux

werden muss.
Über diesen Optionen hinaus wurden die Option *-s* eingeführt. Gerade beim häufigen Anpassen oder Experimentieren sind die Größen der einzelnen Dateien von Interesse. Die Option hat einen optionalen Parameter über den ein Dateiname angegeben werden kann in den die Ergebnis fortlaufend gespeichert werden.
Zuletzt wurde die Option *-h* eingeführt, die eine Auflistung sämtlicher Optionen und deren ausführliche Beschreibungen ausgibt. Darin werden auch besondere Fälle dargelegt und welche obligatorischen und optionalen Parameter die Optionen haben.

Weitere Anpassungen Eine der wichtigsten Änderungen am Basisskript, war die Integration des Messwerkzeugs und weiterer Dateien. Diese wurde vor der Ausführung des Skriptes in einem Sammelordner abgelegt. Wie Auflistung 3.10 belegt, werden diese Dateien dann durch das Skript an die jeweiligen Orte abgelegt. Die Binärdateien des Messwerkzeugs wurden in das *bin*-Verzeichnis kopiert, da sie dadurch von jedem Ort als Programmaufruf ausgeführt werden können. Die Keymap wurde in das Verzeichnis *etc* kopiert, da dies auch dem gewohnten Ordner derartiger Dateien entspricht. Weiter unten wird zudem dargestellt, wie auch die *welcome.txt* in diesem Ordner gespeichert und im Livesystem dann von dort aus auch wieder ausgegeben wird.

Auflistung 3.10: Erweitertes Skript: Integration Testwerkzeug und Keymap[18]

```
1 ...
2 cd _install || exit 1
3 ...
4 # copy FLIs prog
5 cp $CUSTOM_PATH/hardwareBenchmark \
6    $CUSTOM_PATH/ramTest \
7    $CUSTOM_PATH/blockDeviceTest \
8    $CUSTOM_PATH/blockDeviceTest \
9    $CUSTOM_PATH/de.bmp \
10   ./bin/
11 ...
12 cd etc
13 # copy de-keymap to etc
14 cp $CUSTOM_PATH/de.bmp .
15 ...
```

Das upsprüngliche Skript startete nach der Ausgabe der Kernelmeldungen direkt in eine Shell. Dieses Verhalten wurde leicht angepasst. Es wird nun die Nachricht aus der Auflistung 3.11 dargestellte. So weiß der Benutzer sofort, wie das Benchmark zu starten ist.

Auflistung 3.11: Erweitertes Skript: Welcome-Text[19]

```
1 ...
2 # generate welcome message
3 cd etc
4 touch welcome.txt
5 echo '#!/bin/sh' >> welcome.txt
6 echo '  ######################################' >> welcome.txt
```

[18] Quelle: Auszug aus Spi18.
[19] Quelle: Auszug aus Spi18.

```
7 echo ' #                          #' >> welcome.txt
8 echo ' #  Welcome to "DHGE-Benchmark"  #' >> welcome.txt
9 echo ' #                          #' >> welcome.txt
10 echo ' ##########################################' >> welcome.txt
11 echo ' # to start type:              #' >> welcome.txt
12 echo ' # hardwareBenchmark             #' >> welcome.txt
13 echo ' #                          #' >> welcome.txt
14 echo ' ##########################################' >> welcome.txt
15 echo >> welcome.txt
16 cd ..
17 ...
```

Zudem wurde das *init*-Skript angepasst. Die Auflistung 3.12 zeigt, dass über die *loadkeymap* das Tastaturlayout auf *QWERTZ* umgestellt wird. Dafür muss allerdings zuvor die *de.bmp* in das Verzeichnis *etc* kopiert werden.

Auflistung 3.12: Erweitertes Skript: Init[20]

```
1 ...
2 ## generate init
3 echo '#!/bin/sh' > init
4 echo 'dmesg -n 1' >> init
5 echo 'mount -t devtmpfs none /dev' >> init
6 echo 'mount -t proc none /proc' >> init
7 echo 'mount -t sysfs none /sys' >> init
8 echo 'loadkmap < /etc/de.bmp' >> init
9 echo 'clear' >> init
10 echo 'cat /etc/welcome.txt' >> init
11 echo 'setsid cttyhack /bin/sh' >> init
12 ...
```

Anschließend werden alle Nachrichten vom Bildschirm entfernt und die *welcome.txt* ausgegeben und wie zuvor auch die Shell gestartet.

Die Übertragung auf das Medium Um das ISO-Image nutzen zu können, sollte es auf ein Medium übertragen werden. Bei der Verwendung eines optischen Mediums kann zum Beispiel mit den Windows-Boardmitteln das Image gebrannt werden. Andere Brennprogramme wurden nicht getestet, sollten aber das gleiche Ergebnis erzielen.
Für die Übertragung auf einen USB-Stick eignet sich *Rufus*[21]. Die Software wurde für eben diesen Zweck gebaut. Dabei kann auch die Boot-Methode angegeben werden. Dennoch ist zu beachten, dass das Image die Voraussetzungen von *uefi* erfüllen muss. Andernfalls bleibt die Option erfolglos. Im Test wurde ein USB-Stick mit dem Image ausgestattet und für Bios-Boot aufbereitet. Dieser Testlauf verlief erfolgreich.

Durch diese Änderungen konnte das Skript für die Studienarbeit, bzw. für den späteren Gebrauch innerhalb der DHGE angepasst werden und sorgt damit für eine komfortablere

[20] Quelle: Auszug aus Spi18.
[21] https://rufus.ie/de_DE.html

Nutzung als es das Basisskript zuließ. Dennoch bleibt zu erwähnen, dass die Änderungen eben auf Basis der Anforderungen durchgeführt wurden. Damit weichen diese in wenigen Fällen von der gewöhnlichen Nutzung eines Shell-Skriptes ab. So müssen zum Beispiel die Dateien des Hardwaretests und der Keymap im Ordner *custom* abgelegt werden, damit diese auch später in der Liveumgebung genutzt werden können. Dies könnte zu Fehlerverhalten führen, wenn das Skript anderweitig genutzt oder bestimmte Voraussetzungen nicht erfüllt werden, da sich die Anforderungen geändert haben. Somit sollte das erweiterte Skript trotz Dokumentation aufmerksam genutzt werden.

3.3 Der Testlauf

Bis zum finalen Testlauf wurden alle Testläufe in einer virtuellen Maschine durchgeführt. Zum Einsatz für die Virtualisierung kam *VirtualBox*[22] von Oracle. Die Konfiguration der virtuellen Maschine war dabei relativ unwichtig, da das Live-System vor dem Betriebsystem operierte. Dennoch mussten Anpassungen vorgenommen werden. Zum einen durfte das System nicht über EFI starten, da dieses vom Image nicht unterstützt wurde. Zum anderen musste ausreichend Arbeitsspeicher vorhanden sein, da ansonsten der Kernel und das Root-Filesystem nicht in den Speicher passen würden. Da hier mit den Standardeinstellungen für eine Linux-Maschine gearbeitet wurde, fiel der Arbeitsspeicher nicht weiter ins Gewicht. Der Bootmodus konnte durch die in der Abbildung 3.3 dargestellte Option angepasst werden.

Der finale Testlauf wurde im Labor der DHGE durchgeführt. Es wurde eine repräsen-

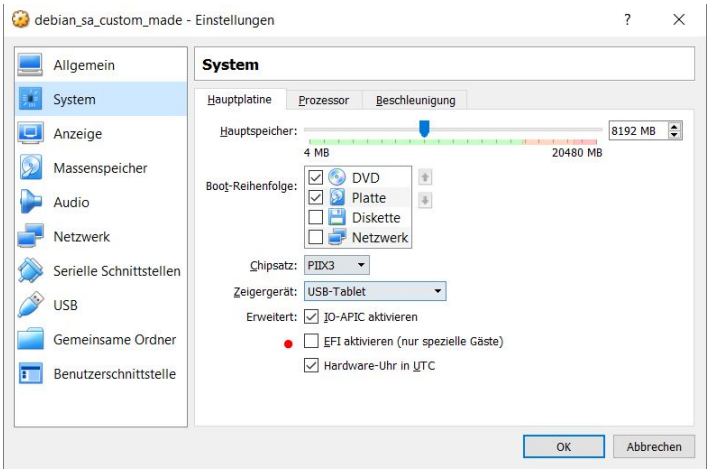

Abbildung 3.3: Bootoption VirtualBox[23]

tative Hardware vom Praktikumsbetreuer zur Verfügung gestellt. Darunter befanden sich auch die Controller, die im Versuch *IT2* zum Einsatz kommen. Weiterhin war der Entwickler des Benchmarks anwesend, um zum einen für Fragen und Anpassung zur Verfügung stehen und zum anderen um selbst das Benchmark zu testen.

[22] https://www.virtualbox.org/
[23] Quelle: selbst erstellte Grafik

3.3.1 Die Vorbereitung

Um das Image repräsentativ nutzen zu können, musste es in zwei Formen vorliegen: auf CD gebrannt und auf einem bootfähigen USB-Stick. Der Brennvorgang auf CD kann durch jede x-beliebige Brennsoftware erfolgen und wurde mit den Boardmitteln von Windows umgesetzt. Das Image wird auf die CD übertragen und beinhaltet bereits alle Inforamtionen die ein Bootmanager benötigt um das Medium als bootfähig zu verstehen. Die Übertragung auf den USB-Stick wurde, wie weiter oben empfohlen, mit der Software *Rufus*[24] umgesetzt. Diese Software ermöglicht es einen USB-Stick in ein bootfähiges Medium umzuwandeln und ein Image auf diesem zum Start zu kopieren.

3.3.2 Die Durchführung

Da zum Zeitpunkt des finalen Testlaufes die UEFI-Unterstützung noch nicht gegeben war, musste im Bios die Bootmethode auf Legacy, bzw Bios umgestellt werden. Anschließend bootete das System problemlos und das Live-System startete wie erwartet. Im Folgenden wurden dann zunächst der Arbeitsspeicher- und CPU-Test durchgeführt. Der Arbeitsspeicher wurde sowohl im Single- als auch Dual-Channel-Mode erfolgreich getestet. Die Konfiguration der CPU wurde mehrfach abgeändert und zeigte weiterhin keine Probleme bei der Ausführung des Benchmarks.
Abschließend wurden die Massenspeichermedien getestet. Dabei wurden verschiedene Typen (HDD und SSD) über jeweils verschiedene Schnittstellen (SAS, SATA, M.2 und USB2.0 und USB3.0) mit dem Testrechner verbunden. Der *blockDeviceTest* lief bei allen durchgeführten Konfigurationen erfolgreich durch. Zudem konnte die Kompatibilität des Kernels mit den Schnittstellen und Medien über den Befehl *df* und *sudo blkid* bestätigt werden. Darauffolgend wurden die Kompatibilität zu den Controllern und *RAID*-Leveln getestet. Dabei wurde jeweils eine Karte in den Rechner eingebaut und mit verschiedenen *RAID*-Leveln getestet. Auch hier verliefen die Tests erfolgreich. Eine Ausnahme stellte die *Adaptec ASR-6805: Microsemi PM8013 Dual Core RAID-on-Chip* dar. Es wurde versucht über einen generellen Treiber für Adaptec-Karten die Kompatibilität zu bewerkstelligen, da es keinen expliziten Treiber im Kernel gab. Dieser Versuch stellte sich als erfolglos dar. Die Information wurde dem Praktikumsbetreuer mitgeteilt, der daraufhin eine Treiber-CD für die Karte aushändigte. Diese enthält auch Treiber für Linux-Systeme. Dennoch konnte die Thematik bisher nicht weiter verfolgt werden.
Im Gesamtbild stellt sich der finale Testlauf als erfolgreich heraus. Im Wesentlichen konnten zwei Anforderungen nicht umgesetzt oder getestet werden. Dazu gehört zum einen die Kompatibilität zur Adaptec-Karte und der Start über *UEFI*-Boot. Darüber hinaus konnten die Anforderungen erfüllt werden.

[24] https://rufus.ie/de_DE.html

3.3.3 Die Nachbereitung

Es wurde im Nachgang versucht den UEFI-Boot zu realisieren. Dabei wurde sich an den Skripten des umfangreicheren Repository der Variante 2[25] orientiert. Insbesondere die Dateien *13_prepare_iso.sh* und *14_generate_iso.sh* wurden einer genaueren Betrachtung und Bearbeitung unterzogen, wobei die Grundstruktur erhalten blieb und nur die Variablen dem erweiterten Skript entsprechend angepasst. Dennoch ließ sich kein funktionsfähiges ISO-Image über die Option *both* und *uefi* erstellen. Die virtuelle Maschine meldete *Error reported: Unsupported*, wie die Abbildung 3.4 links belegt.

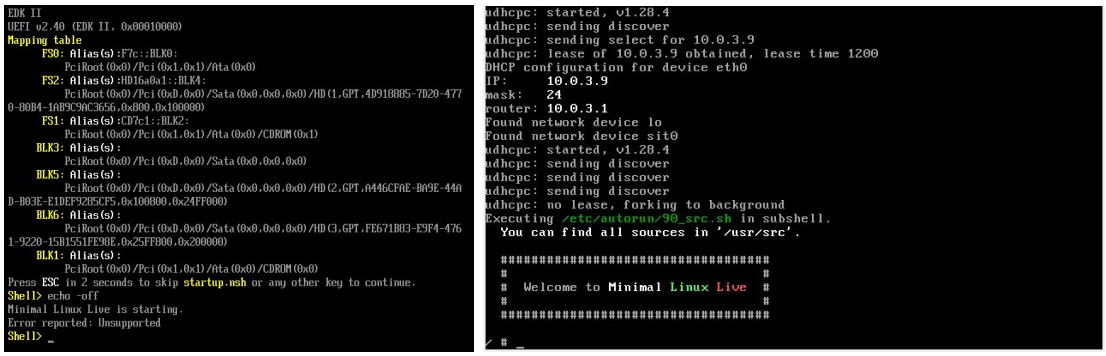

Abbildung 3.4: UEFI-Boot: integrierte Lösung (links) und Originallösung (rechts)[26]

Da ein Treiberproblem vorliegen konnte, wurde das ISO-Image auf eine CD gebrannt und an einer Workstation mit UEFI-Kompatibiliät erneut getestet. Auch bei dieser Konfiguration wurde die CD nicht als UEFI-bootfähig erkannt. Der Legacy-Boot konnte hingegen durchgeführt werden. Zuletzt wurde versucht ein UEFI-Image mittels des unbearbeiteten Quellcodes zu erstellen, um zu prüfen, ob der Fehler an den Anpassungen lag. Das entstandene Image konnte sowohl über UEFI- als auch BIOS-Boot erfolgreich gestartet werden, wie Abbildung 3.4 rechts belegt. Somit liegt der Fehler an der mangelhaften Integration in das erweiterte Skript. Für eine weitere Bearbeitung dieser Thematik fehlte jedoch am Ende die Zeit.
Darüber hinaus wurde die Treiber CD des Adaptec-Controller begutachtet. Auf dieser befanden sich Treiber verschiedenster Systeme, unter anderem auch für Linux. Jedoch lässt sich kein passender Treiber zum aktuell verwendeten Linux-Kernel finden. Anschließend wurde auf der Homepage des Herstellers gesucht, aber auch hier gab es nur Treiber, die bereits seit 2 Jahren nicht weiter bearbeitet wurden. Entsprechend dieser Sachlage, wurde sich gegen den Treiber entschieden und für ein stabiles Image.

[25] https://github.com/ivandavidov/minimal
[26] Quelle: selbst erstellte Grafik

4 Auswertung und Ausblick

Die Ausarbeitung und der Testlauf sind abgeschlossen und es verbleibt der Fazit. Im Folgenden wird eine kritische Nachbetrachtung der Arbeit vorgenommen und der Ablauf reflektiert. Darauf folgt eine Auslistung der Verbesserungsmöglichkeiten.

4.1 Kritische Nachbetrachtung

Die Arbeit konnte im Kern umgesetzt werden und bearbeitet die Thematik im hinreichenden Ausmaß. Im ersten Kapitel folgte nach einer kurzen Einstimmung auf das Thema die Problemstellung und die daraus abgeleitete Zielstellung sowie Abgrenzung. Das Kapitel endete mit der Vorgehensweise innerhalb der Studienarbeit.
Das zweite Kapitel widmete sich der theoretischen Vorbetrachtung und der Vorbereitung auf die praktische Durchführung. Dabei wurde als erstes der generelle Startvorgang eines Computers detailliert dargelegt. Dem schloss sich die Darlegung der Anforderungen an, wobei vorweg das Praktikum der Studierenden im vierten Semester beschrieben und als initialer Grund angeführt wurde. Aus diesem wurden die obligatorischen und optionalen Kriterien abgeleitet, wie auch Punkte, die nicht in das Ergebnis einfließen sollten. Zuletzt wurde das Werkzeug *Minimal-Linux-Live* kurz vorgestellt und ein kurzer Abriss über alternative Möglichkeiten der Erstellung gegeben.
Die Realisierung war der rote Faden des dritten Kapitel. Zunächst wurde das Werkzeug ausführlich beschrieben und die Vorgehensweise des Build-Skriptes erörtert. Dann folgte der Kernpunkt der Arbeit, die manuelle Anpassung des ursprünglichen Skriptes an die Anforderungen. Dabei wurde dargelegt, wie die Kernelkonfiguration angepasst werden kann und wo dies getan wurde. Die Integration, die Kurzbeschreibung und die Probleme bei der Einführung des Messwerkzeugs stellten den nächsten Punkte dar auf den die Anpassung des Shell-Skriptes, bzw. die Integration von Optionen in dieses folgte. Das Kapitel schloss mit der Durchführung, der Auswertung und der Nachbereitung des Testlaufes ab.
Die Studienarbeit konnte in ihrem Kern umgesetzt werden und bis auf zwei wurden alle obligatorischen Anforderungen erfüllt. Die Abgrenzungen und unnötigen Funktionen wurden berücksichtigt und durch die Anpassungen in der Kernelkonfiguration weitreichend umgesetzt. Dennoch konnten nicht alle obligatorischen Anforderungen realisiert werden. Allein die erfolglose Umsetzung der UEFI-Boot-Methode stellte eine nicht zu unterschätzende lehrreiche Herausforderung dar. Interessant wäre es gewesen zu erfahren, ob die Umsetzung dieses Ziels durch die Nutzung des zweiten Repository anstelle des erstens geglückt wäre. Es liegt die Vermutung nah, dass die Einarbeitung in das zweite wesentlich komplexere Repository zu viel Zeit gekostet hätte und somit eine mangelhafte

Umsetzung erfolgt wäre. Dennoch waren die Einblicke über die Organisation derartige Aufgaben bei anderen Personen aufschlussreich.

4.2 Verbesserungsmöglichkeiten

Der Studienarbeit ist ein rundes Ergebnis entsprungen. Das Image kann stabil und zuverlässig über das angepasste Build-Skript und ohne viel Aufwand erstellt werden. Die Übertragung auf ein Medium ist beschrieben. Nebst kleineren Aufgaben sollte als nächstes die Realisierung des UEFI- oder des *dualen* Boots in Angriff genommen werden. Dadurch würden weitere unnötige Schritte entfallen. Abseits dessen sollte die Struktur des Build-Skriptes überarbeitet werden. Gerade die Arbeit mit dem komplexeren Repository am Ende zeigte eine wohlüberlegte Struktur, die gut nachvollziehbar und vor allem auch modular ausführbar war. Die Umsetzung der Konfigurationsanpassungen durch separat abgelegte *.config*-Dateien wäre ebenso denkbar, wenngleich das Menü durch die Hilfefunktion eine informative Alternative offeriert.
Wie so oft, kann auch in dieses Projekt noch sehr viel Zeit investiert werden, um es immer weiter zur Perfektion zu treiben. *Doch manchmal ist gute eben besser als perfekt.*

Literatur

[Bel95] Bell, A.: *The ISO 9660 File System*, 1995, `http://users.telenet.be/it3.consultants.bvba/handouts/ISO9960.html` Abruf: 12. 11. 2018

[Dav18] Davidov, I.: *About This Project*, 2018, `http://minimal.linux-bg.org/#about` Abruf: 14. 11. 2018

[Dua08] Duarte, G.: *The Kernel Boot Process*, 2008, `https://manybutfinite.com/post/kernel-boot-process/` Abruf: 12. 11. 2018

[Gri18a] Grimm, A.: „Praktikum Informationstechnik, Versuch IT1“, auf der CD: Gri18a_Versuch1(Rechneraufbau).pdf, 2018

[Gri18b] Grimm, A.: „Praktikum Informationstechnik, Versuch IT2“, auf der CD: Gri18b_Versuch2(Speicher+Raid).pdf, 2018

[Gri18c] Grimm, A.: „Praktikum Informationstechnik, Versuch IT3“, auf der CD: Gri18a_Versuch3(Tuning+Sicherheit).pdf, 2018

[Her06] Hertzog, U.: *Linux: kompakt, komplett, kompetent*, München, 2006

[int08] intel: *EFI Shells and Scripting*, 2008, `https://software.intel.com/en-us/articles/efi-shells-and-scripting` Abruf: 11. 11. 2018

[Koz01] Kozierok, C. M.: *System BIOS*, 2001, `http://www.pcguide.com/ref/mbsys/bios/index.htm` Abruf: 10. 11. 2018

[Pol11] Pollard, J. d. B.: *The EFI boot process.* 2011, `https://jdebp.eu/FGA/efi-boot-process.html` Abruf: 10. 11. 2018

[Sch16a] Schmitt, T.: „xorriso“, Aus der MAN-Page zu xorriso, 2016

[Sch16b] Schmitt, T.: „xorrisofs“, Aus der MAN-Page zu xorrisofs, 2016

[Spi18] Spillner, R.: „Erweitertes Build-Skript“, Auf der CD unter Build:build_mll_custom.sh, 2018

[ubu17] ubuntu Deutschland e. V.: *UDF*, 2017, `https://wiki.ubuntuusers.de/UDF/` Abruf: 12. 11. 2018

[ubu18] ubuntu Deutschland e. V.: *Bootvorgang*, 2018,
`https://wiki.ubuntuusers.de/Bootvorgang/`
Abruf: 10. 11. 2018

[Wil14] Williamson, A.: *UEFI boot: how does that actually work, then?*, 2014,
`https://www.happyassassin.net/2014/01/25/uefi-boot-how-does-that-actually-work-then/`
Abruf: 10. 11. 2018

[Yoc18] Yocto Project: *Yocto Project Quick Build*, 2018,
`https://www.yoctoproject.org/docs/2.5.1/brief-yoctoprojectqs/brief-yoctoprojectqs.html`
Abruf: 13. 11. 2018

Anhang

A.1 Zusammenfassung Anforderungen

Stand: 12.11.2018
Die vorliegenden Anforderungen wurden aus den Inhalten der Besprechungen zusammengetragen.

A.1.1 Obligatorische Anforderungen

- Priorität liegt bei Geschwindigkeit und nicht bei Größe
- Startvorgang unterhalb von 10 Sekunden
- Größe des Images kleiner als 30 MB
- Oberfläche: interaktiv, text-basierend
- Architektur: 64bit
- CPU: x86, Intel und AMD
- BOOT: UEFI, alternativ Bios
- Kernel
  - statisch gelinkt
  - möglichst klein, bzw. entschlackt
  - keine Moduleloading
- Treiber
  - HDD / SSD: SAS, SATA, USB (v1,v2,v3), RAID-Controller
  - Visueller Output: VGA
  - SAS-Controller: LSISAS3445E-R : LSISAS1068E
  - RAID-Controller
    - Darwin Control DC-324e RAID: Maevell 88SX7042
    - LSI 3ware SAS 9750-4i: I/O Processor/SAS Controller: LSISAS2108 RAID-on-Chip (ROC)
    - Adaptec ASR-6805: Microsemi PM8013 Dual Core RAID-on-Chip
- Testlauf
  - Alle Raid-Controller inklusive RAID-Konfigurationen
  - Alle Schnittsellen

A.1.2 Optionale Anforderungen

- Init-System möglich
- Sowohl UEFI als auch Legacy BOOT möglich
- Publizieren verschiedener .configs für den Kernel, die per Patch eingefügt werden
- unterschiedliche Bootmenüeinträge
  - Starte in interaktive Oberfläche: Shell
  - Starte in Benchmark und kehre Shell¬ zurück nach Abschluss

A.1.3 Abgrenzung und unnötige Funktionen

- Netzwerkunterstützung
- keine graphische Oberfläche
- Secure Uefi

dotSource®

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich,

1. dass ich meine Studienarbeit mit dem Thema:
   Software- und Hardwaretest:Minimal Linux
   ohne fremde Hilfe angefertigt habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und

3. dass ich meine Studienarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Jena, den 27. Mai 2019

[redacted]