TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Optimalizace startu OS Linux

Bakalářská práce

Studijní program: B0613A140005 Informační technologie
Studijní obor: Aplikovaná informatika

Autor práce: **David Dlouhý**
Vedoucí práce: Ing. Lenka Kosková Třísková, Ph.D.
Ústav nových technologií a aplikované informatiky

Liberec 2022

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání bakalářské práce

Optimalizace startu OS Linux

Jméno a příjmení: **David Dlouhý**
Osobní číslo: M19000010
Studijní program: B0613A140005 Informační technologie
Specializace: Aplikovaná informatika
Zadávající katedra: Ústav nových technologií a aplikované informatiky
Akademický rok: **2021/2022**

Zásady pro vypracování:

1. Seznamte se s procesem startu OS Linux a popište jeho základní fáze startu sytému.
2. Analyzujte časovou náročnost jednotlivých fází startu OS Linux.
3. Navrhněte postup pro zrychlení startu operačního systému úpravou konfigurace již existující verze jádra (vhodné nastavení startu systémových služeb a procesů).
4. Navrhněte postup pro zrychlení startu operačního systému úpravou konfigurace jádra OS před jeho překladem (volba modulů a funkcí jádra).
5. Ověřte navržené postupy na reálném zařízení (jednodeskový počítač s architekturou ARM) a změřte rozdíl startu optmializovaného a neoptimalizovaného systému.
6. Výsledky přehledně shrňte a vyberte nejvhodnější postup.

Rozsah grafických prací:	dle potřeby dokumentace
Rozsah pracovní zprávy:	30-40 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] Love, R.: Linux Kernel Development 3rd ed., Addisons Wesley, 2010, ISBN: 0-672-32946-3.
[2] Simmonds, Ch: Mastering Embedded Linux Development, Packt Publishing, Ltd., Birmingham 2015, ISBN: 978-1-78439-253-6.
[3] SALVADOR, Otavio a Daiane ANGOLINI. Embedded Linux development with Yocto project: develop fascinating Linux-based projects using the groundbreaking Yocto project tools. Birmingham: Packt Publishing, 2014. Community experience distilled. ISBN 978-1-78328-233-3.
[4] Babar Yogesh: Hands-on Booting: Learn the Boot Process of Linux, Windows, and Unix. Apress, 1. 2020, ISBN: 1484258894.
[5] Ríos, A. L.: Linux Driver Development for Embedded Processors – Second Edition: Learn to develop Linux embedded drivers with kernel 4.9 LTS, Independently Published, 2018, ISBN: 1729321828.

Vedoucí práce:	Ing. Lenka Kosková Třísková, Ph.D. Ústav nových technologií a aplikované informatiky
Datum zadání práce:	12. října 2021
Předpokládaný termín odevzdání:	16. května 2022

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 19. října 2021

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

Datum: 13.05.2022

Podpis:

Poděkování

Tímto bych chtěl poděkovat Ing. Lence Koskové Třískové, Ph.D. za vedení práce, odborné konzultace a za čas, který mi ochotně věnovala vždy, když bylo třeba. Rád bych poděkoval i rodinnému kruhu za podporu a pochopení. Dále pak děkuji Mgr. Heleně Jandové.

Abstrakt

Tato práce popisuje start operačního systému Linux a také jeho optimalizaci. Ukazuje na problémová úzká místa startu. Uvedeny jsou hypotézy optimalizací zaměřené zejména na optimalizaci služeb operačního systému. Dále pak hypotézy zaměřené na konfiguraci jádra Linux a na volbu modulů tohoto jádra. Uvedena je i práce s projektem Yocto, a proto je právě do hypotéz zahrnuta i optimalizace vrstvy meta-raspberrypi. V práci je popsána praktická aplikace na jednodeskovém PC Raspberry Pi 3B+. Tyto výsledky jsou přehledně shrnuty a práce obsahuje modelovou situaci s výběrem vhodné optimalizace. Výsledky práce jsou založeny na naměřených datech, které jsou volně k dispozici. Práce také popisuje postup měření dat startu a následné zpracování těchto veřejnosti přístupných dat. Řešeno je taktéž i uchování těchto dat ve vhodném formátu pro strojové zpracování. Obsahem práce je taktéž návrh a využití ekosystému pro optimalizaci, měření a následnou analýzu dat. Veškeré optimalizace jsou řešeny atomicky a následně v různých kombinacích. Využitelnost práce je podpořena doplňující dokumentací, a to zejména z důvodu předem avizovaného využití této práce ve výzkumném projektu, který je na Technické univerzitě v Liberci řešen. Shrnutí porovnává jednotlivé optimalizace a volí výslednou optimalizaci dle modelové situace. Vizualizuje naměřená data pro realizované hypotézy. Čtenář se tedy dozví, jak operační systém Linux startuje, kde se nacházejí problémová úzká místa. Následně se seznámí s procesem optimalizace a s klíčovým měřením dat. Dozví se také, jakým způsobme je tato data vhodné zpracovat a jaké hodnoty je záhodno sledovat. Závěrem je pak uživatel seznámen s postupem volby výsledné optimalizace na základě konkrétní modelové situaci.

Klíčová slova

Linux, optimalizace startu, Yocto, Systemd, Raspberry Pi

Abstract

This thesis describes the startup sequence of the Linux operating system and also its optimization. It points out the problem bottlenecks of the boot process. Optimization hypotheses are given, mainly focused on optimizing the operating system services. Furthermore, hypotheses focusing on the configuration of the Linux kernel and the choice of modules of this kernel are presented. Work with the Yocto project is also presented, and it is for this reason that the optimization of the meta-raspberrypi layer is included in the hypotheses. A practical application on a Raspberry Pi 3B+ single board PC is described. These results are clearly summarized and the thesis includes a model situation with the selection of an appropriate optimization solution. The results of the work are based on measured da-data that are freely available. The thesis also describes the procedure for measuring the launch data and the subsequent processing of this publicly available data. The storage of these data in a suitable format for machine processing is also addressed. The design and use of an ecosystem for optimization, measurement and subsequent data analysis is also included. All optimizations are solved atomically and then in various combinations. The usefulness of the project is supported by additional documentation, especially because of the previously announced use of this project in a research project that is being conducted at the Technical University of Liberec. The summary compares the individual optimizations and selects the resulting optimization according to the model situation. It visualizes the measured data for the implemented hypotheses. The reader thus learns how the Linux operating system starts, where the problem bottlenecks are located. Then he will learn about the optimization process and the key data measurement. He will also learn how to process the data and which values are useful to monitor. Finally, the user is then introduced to the process of selecting the resulting optimization based on the specific model situation.

Keywords

Linux, boot optimization, Yocto, Systemd, Raspberry Pi

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

API	rozhraní pro programování aplikací
ARM	architektura procesorů
ASCII	standard pro definici znaků v informatice
BIOS	firmware, který poskytuje komunikaci s HW
BP	bakalářská práce
BSD	licence pro svobodný software
BT	Bluetooth
CD	automatizovaný proces nasazení kódu; typ paměťového média
CI	automatizovaný proces integrace kódu
CPU	procesor
DB	databáze
DHCP	protokol umožňující dynamické přidělování IP adres
DNS	protokol zajišťující překlad názvů domén na IP adresy
ECDH	Dieffieho-Hellmanův šifrovací protokl s využitím eliptických křivek
EXT3	žurnálovací systém souborů
EXT4	žurnálovací systém souborů
FLASH	nonvolatilní elektronicky program. paměť s libovolným přístupem
FS	systém souborů
FTP	protokol a služba pro přenos souborů po síti
GNU	nekompletní počítačový svobodný operační systém projektu GNU
GPU	grafický procesor
GRUB	zavaděč
GUI	grafické uživatelské rozhraní
Gzip	výchozí komprese jádra Linux
HDD	elektromechanické zařízení pro záznam a čtení dat
HTML	značkovací jazyk pro tvorbu webových stránek
HW	hardware
I/O	vstupně/výstupní
ID	identifikační číslo
IP	internetový protokol
IPv4	internetový protokol čtvrté verze
IPv6	internetový protokol šesté verze
JSON	multiplatformní způsob zápisu dat
LAN	lokální síť
LILO	zavaděč
LVM	dynamická správa diskových oddílů
LZO	typ komprese užité pro komprimování jádra Linux
MBR	hlavní zaváděcí záznam v prvním sektoru disku
NoSQL	databázový koncept
OS	operační systém
PAN	osobní síť

PC	osobní počítač
PID	identifikační číslo procesu
POSIX	standard unixových OS
POST	sekvence diagnostických testů prováděná při startu PC
QR	kódy rychlé reakce
RAID	diskové pole
RAM	volatilní paměť s náhodným přístupem
RF	radiofrekvenční
ROM	nonvolatilní paměť, obsah je zapsán výrobcem a je určen pouze pro čtení
RPI	Raspberry Pi
SATA	sběrnice pro připojení HDD a SSD disků
SCP	protokol pro bezpečný přenos dat
SD	typ paměťové karty
SDHC	paměťové médium nahrazující SD
SDRAM	synchronní dynamická paměť RAM
SSD	elektronické zařízení pro záznam a čtení dat
SSH	program a zabezpečený protokol pro komunikaci v počítačové síti
SW	software
SWAP	místo v externí paměti pro odkládání dat z operační paměti
TUL	Technická univerzita v Liberci
UART	PC sběrnice sloužící k sériovému asynchronnímu přenosu dat
UEFI	evoluční náhrada firmware BIOS s absencí vazby na platformu x86-64
URL	jednotný lokátor zdroje, adresa
USB	univerzální sériová sběrnice, rozhraní pro připojení přídavného HW
VFS	virtuální souborový systém
Vim	solidní textový editor
VPS	virtuální privátní server
WLAN	bezdrátová lokální síť
XML	značkovací jazyk pro popis dat
YML	formát čitenlého souboru pro člověka určený k serializaci

Úvod

V této práci popisuji optimalizaci startu operačního systému GNU/Linux. Čtenáři jsem se pokusil detailně přiblížit proces startu operačního systému. Na základě tohoto popisu uvádím možné hypotézy zrychlení startu OS Linux. Popisuji také postup, jak jsem jednotlivé optimalizace prakticky nasadil, jak jsem efektivně realizoval sběr dat, změřil je, a jak jsem tato data analyzoval. Uvedl jsem také i shrnující komparaci jednotlivých optimalizací s výsledným výběrem nejlepší optimalizace dle modelové situace.

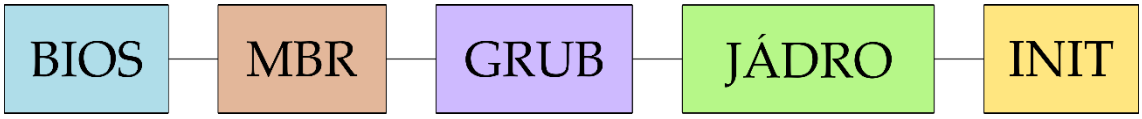
Práce je inspirována reálnými potřebami. V oblasti automotive rostou požadavky na zařízení, která používají odlehčený vestavný operační systém a současně na zařízení, která nastartují co nejdříve, například po otočení klíčku zapalování. Reálně však mnoho operačních systémů nabíhá i déle jak 30 s.

Pro přehlednost jsem jako součást práce vytvořil online rozcestích, který referuje na jednotlivé součásti práce. Tyto součásti jsou dále v práci přiblíženy podrobněji a díky rozcestníku si čtenář může jednotlivé komponenty prohlédnou. Rozcestník je přístupný přes URL adresu http://dlouhybp.tkkzmj.cz či pomocí níže uvedeného QR kódu.

Obrázek 1: QR kód odkazující na stránku dlouhybp.tkkzmj.cz

1 Proces startu OS Linux

Na úvod bylo třeba, abych se seznámil s procesem startu operačního systému Linux. Jedině tak jsem mohl vyvodit hypotézy optimalizací a ty pak prakticky vyzkoušet a změřit jejich dopad na systém. V této kapitole popíšu jednotlivé fáze startů a jejich časovou náročnost.



Obrázek 2: Vizualizace startu OS Linux

Zkráceně a obecně start probíhá následující formou. Stiskneme spouštěcí tlačítko. Dojde k inicializaci firmware nazvaného BIOS. Provede se kontrola HW, a pokud je vše dle požadavků v pořádku, spustí se zavaděč. Pokud se naskytne problém s vadnou pamětí, chybějícím diskem a podobně, začne se signalizovat tento nedostatek. Pokud je vše, jak má, BIOS načte MBR , který načte a zavede startovací jádro. To si řídí spouštění startovacích skriptů, služeb a démonů. Nicméně zejména pozdní fáze mohou mít odlišnou podobu v různých distribucích operačního systému. Například můžeme spouštět systém pomocí konzervativních skriptů a init nebo pomocí moderního démona Systemd. V následujících podkapitolách jsem detailně popsal jednotlivé fáze startu OS Linux.

1.1 BIOS

Tato fáze je univerzální pro různé operační systémy. Uživateli je prezentována v podobě informačních výpisů či vykresleného loga. Existují i varianty s černou obrazovkou. Nastane ve chvíli, kdy stiskneme tlačítko a k jednotlivým komponentám přivedeme napájení. Díky tomu se spustí firmware, který je napsán ve strojovém jazyce a je uložen v paměti na desce. Nejčastěji se jedná o paměť typu FLASH . Dříve se používaly paměti ROM, ale ty se nedají přepisovat. Tím bychom nemohli v případě potřeby firmware aktualizovat. Firmware uložený na desce se nazývá BIOS. Jeho úkolem je provést POST. Jedná se o otestování provozuschopnosti systému. Firmware má definovaný seznam komponent, které musí zkontrolovat, a v případě chyby musí tyto problémy vhodným

způsobem zachytit a signalizovat. Mezi důležité kontroly se řadí: test procesoru, operační paměti a grafické karty. Následuje testování pevných disků a osadních SATA či USB zařízení. Existuje možnost POST přeskočit funkcí quick-boot.

V případě, že jsou klíčové komponenty funkční, dojde k načtení zaváděcího programu z paměti, ve které se nachází BIOS známýtaktéž pod pojmem bootstrap loader. Cílem programu je dle nastavené sekvence prohledávání načtení a spuštění zavaděče. Ten se nachází v bootovacím sektoru pevného disku, CD, diskety nebo jiného média. Konkrétně je to první sektor disku o velikosti 512 bajtů, ve kterém se nachází zaváděcí program (zavaděč / bootloader). Tomuto sektoru se říká Master Boot Record, zkráceně MBR. V případě, že je zavaděč detekován, dojde k jeho načtení do paměti a následně BIOS předá kontrolu nad systémem zavaděči [1].

1.2 Zavaděč systému

Druhý krok startu je v režii zavaděče (bootloader). Nachází se v 1. sektoru zaváděcího disku. Typicky bychom jej hledali v /dev/sda. Pokud máme firmware, který je připravený na bootování ze sítě, je možné využít i tuto možnost. Existují ale i jiná umístění. Toto umístění označujeme jako Master Boot Record (MBR). Obsah MBR je 512 bajtů velký. Prvních 446 bajtů nese informaci o primárním zavaděči, následuje 64 bajtů reprezentujících informaci o tabulce oddílů a na závěr jsou přidány 2 bajty pro kontrolu ověření. Jeho zodpovědností je načtení buď sofistikovanějšího zavaděče, jako je například GRUB, nebo vzácně přímé zavedení systému. Výhodou zavaděče GRUB je možnost snadné ruční konfigurace a interaktivní konfigurace při startu. Taktéž se dá mezi výhody zařadit i konfigurace jádra OS Linux. Mezi další zavaděče se řadí i LILO , které se primárně využívá u serverových instancí. LILO totiž nabízí snadnou instalaci RAID či možnost jednorázového zadání příkazu pro nadcházející start jádra OS. Využívají jej spíše starší systémy a ty novější využívají v drtivé většině již zmíněný GRUB, konkrétně GRUB 2. Dalším příkladem je oblast vestavných operačních systémů, kde se využívá také Das U-Boot.

Jakmile se spustí sofistikovanější zavaděč systému, který získal kontrolu nad systémem od programu BIOS, dojde ke startu dvou po sobě jdoucích fází zavedení. Zejména u desktopových nasazení se často uživatel setká s interaktivním menu. Zde si

můžeme vybrat bootovací konfiguraci včetně OS, který má být spuštěn. Vývojáři taktéž mysleli na odstranění nutnosti potvrdit výběr. Proto se při startu toto menu zobrazuje po určitý časový interval v jednotkách vteřin. Následně se toto menu ukončí a pokračuje se s výchozím nastavením a výběrem. I tak se tu ale objevuje zpoždění v podobě časového intervalu.

Fáze č. 1 označovaná jako Stage 1 nedokáže pracovat se systémem souborů. Pouze referuje na místo na disku, kde je uložen zavaděč pro fázi 2 (Stage 2) a ten spustí. Úkolem 2. fáze je nahrání jádra do paměti. Provede se předání parametrů, kontrola a následná inicializace jádra. V případě, že využíváme UEFI, zavaděč není potřeba a datový přenos linuxového jádra je uskutečněn napřímo. Vždy je ale primárním úkolem zkopírovat do paměti startovací kód OS a poté jej spustit [1].

1.3 Inicializace jádra

Ve fázi označené jako inicializace jádra je hlavním cílem nejprve načtení jádra do paměti. Konkrétně se načítá komprimovaný obraz jádra. Formát komprimovaného jádra je většinou zImage či bzImage. Nástroje pro práci s tímto formátem jsou k dispozici v knihovně zlib. Jádro musí samo sebe před použitím dekomprimovat z tohoto formátu. Nutností je zavedení základní správy paměti a pomocných nástrojů pro tyto a následující úkony. Díky metodě Head se provede základní nastavení jádra a následně se kompletně dekomprimuje soubor s obrazem jádra do fyzické paměti stroje. Tato část paměti není přímo mapovatelná tabulkou stránek.

Po načtení se pokračuje spuštěním jádra. Vytvoří se stránkovací tabulky pro stránkování paměti (swapper). Jádro také musí při spuštění detekovat typ procesoru včetně jiných komponent, jako je například matematický koprocesor. Jakmile je identifikace hotová a došlo k nastavení nezbytného HW, stránkování paměti a nastavení systémových funkcí, tak se přepne na samotnou funkcionalitu jádra, která napřímo závislá na konkrétní architektuře CPU není. Zde končí činnost hlavního spouštěče jádra. Tento proces je realizován funkcí start_kernel(), kdy je vykonána inicializace v podobě provádění velkého množství inicializačních funkcí. Tato funkce provede množinu systémových konfigurací. Konkrétně se jedná o přípravu a konfiguraci přerušovacích zařízení, konfiguraci paměti a spuštění inicializačního procesu. Taktéž je nutností připojit

inicializační RAM disk (initrd). Ten se v zaváděcí fázi načetl jako dočasný kořenový systém souborů. Inicializační démon (inid) u moderních linuxových jader dokáže přímo z paměti načíst moduly jádra a jiné potřebné ovladač. Pokud bychom načítaly moduly a ovladače z externí paměti (pevný disk aj.), museli bychom spoléhat na korektní připojení a chod jiných zařízení. Navíc ovladač SATA by se tak jako tak musel načíst právě z operační paměti. S tou externí by jádro nevědělo, jakým způsobem pracovat = nemělo by potřebné nástroje = nemělo by ovladač SATA. Ovladače a moduly taktéž můžeme napřímo přeložit do samotného jádra. Jedná se o statické zavedení, které lze měnit pouze v konfiguračním nástroji jádra. Alternativou jsou pak právě moduly a ovladače načtené inicializačním démonem a jádro tak může mít menší rozměr. Zároveň je taktéž potřeba inicializovat virtuální zařízení, jako jsou LVM disková pole RAID a podobně. Nutností je taktéž volání funkce pivot_root(). Ta přepne systémy souborů. Respektive odpojí dočasný kořenový systém souborů určený pouze pro čtení a místo něj připojí skutečný systém souborů uložený právě v externí paměti. S povolenými přerušeními muže převzít kontrolu nad celým řízením systémů a procesů plánovač. Taktéž se zavede knihovna GNU C.

Výsledkem této fáze je inicializace registrů procesoru a kontrola jeho typu, inicializace datových struktur a systémové konzole, podpora zavedení nestatických modulů, VFS, napojení vyrovnávací cache paměti a funkční procesorová komunikace. Dostáváme plně funkční jádro, které ovládá všechny systémové procesy, spravuje paměť, plánuje procesy, stará se o vstupně/výstupní zařízení a o meziprocesorovou komunikaci. Jádro tak celkově řídí systém a skrze inicializační proces Init nastaví uživatelský prostor a procesy potřebné právě pro uživatelské prostředí. Taktéž se jádro stará o zprostředkování HW stroje pro SW. Mezi standardní funkce jádra se taktéž řadí multitasking, alokace, přístup a správa paměti a odkládání do prostoru SWAP . Uživateli je inicializace jádra prezentována například jako série rychlých výpisů, které zmizí. Existuje i konfigurace, kdy čekáme na uživatelův pokyn pokračovat, což je vhodné při ladění systému. Nicméně tyto výpisy je možné získat příkazem dmesg. U systému s potřebou maximální optimalizace se tyto výpisy vůbec neprovádějí [2].

1.4 Inicializační proces

Poslední fází stratu je proces nazvaný Init. Jeho umístění je většinou v /sbin/init, ale toto tvrzení je relevantní pouze pro konkrétní distribuce OS Linux. Init je prvním jádrem spuštěným procesem. Jedná se o první proces spuštěný v uživatelském prostoru. Identifikační číslo tohoto procesu (PID) je vždy 1. Init proces zavádí uživatelské prostředí a pracuje pouze v uživatelském prostoru s podporou volání systému. Taktéž je nutné spustit úlohu nečinnost cpu_idle(). Služba zaštiťující tento první proces se nazývá Systemd. Dříve se používal System V. Jako alternativu je možné taktéž využít BSD init, který primárně cílí na vysokou rychlost. Jeho nevýhodou může být nutnost větší obezřetnosti u méně přehledné konfigurace. V případě, že je proces init nakonfigurován chybně, start OS neproběhne úspěšně.

Úkolem procesu init je zajištění funkčního systému pro běžné potřeby a použití. Provádí se kontrola systému souborů, spouští se jednotliví démoni nebo se hledají závislosti jednotlivých modulů jádra. Realizace pomocí System V, BSD či Runit využívá širokou škálu startovacích a spouštěcích skriptů spouštěných konzolí Shell. Služby jako Systemd nebo Upstarter využívají jiný přístup, a to že poskytnou uživateli konfiguraci skrze konfigurační sobory a následně při dalším startu tuto konfiguraci předávají svým binárním komponentám.

V init procesu je třeba zajistit si běh jednotlivých služeb a démonů. Tyto služby a démoni se umisťují do několika úrovní či cílů. V praxi si například server spustí webový server, systém řízení báze dat (databázový server) a službu FTP či samotnou síť. Služby a programy jsou spouštěny v jednotlivých úrovních, kterým se říká runlevels. Tyto úrovně běhu používá spíše modernější přístupy jako například Systemd. Ten tedy spustí programy v jednotlivých úrovních běhu očíslovaných od 0 do 6 s dodatkovým nožovým cílem. Úroveň 0 představuje cíl zastavení, 1 je rezervována pro režim jednoho uživatele, 2. úroveň reprezentuje režim pro více uživatelů bez NFS, víceuživatelský systém pak využívá třetí úroveň. 4. úroveň není využita. X11 reprezentuje úroveň č. 5 a cíl restart nalezneme na úrovni s číslem 6. Taktéž je možné využít inicializační tabulku inittab. Postupně se tak uvádí OS Linux do použitelného stavu. Připojí se systémy souborů, které

jsou definovány souborem /etc/fstab a to včetně speciálních odkládacích souborů a oddílů. Systemd také prochází pozůstatky SysV z důvodu zpětné kompatibility.

Uživateli se tato fáze vizualizuje v podobě výpisů startů jednotlivých služeb a démonůči jiných programů a pomocných výpisů. Po vykonání této fáze konečně vidí přihlašovací dialog. V případě, že systém vypínáme doporučeným postupem, voláme init s požadavkem na zavření všech funkcí uživatelského prostoru. Init se následně ukončí a jádro provede řízené vypnutí počítače [1].

2 Možnosti zrychlení startu

Na základě znalosti startu OS Linux jsem mohl vyvodit několik hypotéz ohledně optimalizace startu. Při průchodu jednotlivými fázemi stratu jsem našel řadu míst, která se dají optimalizovat. V této kapitole jsem je zmínil a dále v práci popsal realizaci a výsledek vybraných optimalizací.

Moje první myšlenky směřovaly k HW zařízení. HW může pro systém znamenat pevnou hranici, která nám z technologického hlediska již více nedovolí optimalizovat. Proto je vhodné co nejlépe zvolit HW konfiguraci dopředu. Případně některé komponenty nahradit za jiné. Příkladem může být například změna pevného disku typu HDD na typ SSD. Fáze, kdy se spouští BIOS, se prakticky optimalizovat nedá a je dána výrobcem desky a dodavatelem firmware BIOS. V případě, kdybych zde měnil firmware, získal bych v lepším případě velmi malou časovou optimalizace za cenu obětovaného času pro studium tvorby firmware BIOS. Obdobně je na tom MBR, kdy bych musel zefektivnit strukturu toho záznamu pro rychlé čtení.

Jako první atraktivní a zadáním vyžadovaná optimalizace se mi nabízela konfigurace jádra obnášející i vhodnou volbu modulů. Jádro Linux totiž nabízí řadu konfigurací, které na správném HW se správným scénářem užití může přinést časovou úsporu. Volbou modulů bych mohl zajistit napřímo velikost jádra a tím i zmenšit čas přenosu jádra do paměti nepřímo pak ovlivnit poslední fázi startu OS, a to inicializační skript, respektive spouštěné služby. Tyto služby start nejvíce prodlužují a platí přímá úměrnost mezi počtem služeb při staru a časem startu OS. V případě, kdybych zařízení používal ke konkrétnímu účelu, instaloval bych do systému opravdu jen nutné závislosti. OS by pak poskytoval jen nutné služby pro konkrétní aplikaci či nástroje nutné pro správu zařízení. Optimalizovat se dají i jednotlivé skripty volané při startu systému, a to zejména změnou syntaxe. Možností je taktéž i systém bez grafického uživatelského rozhraní, v opačném případě by se mezi optimalizace řadila i eliminace tohoto typu uživatelského rozhraní [2].

Vždy ale záleží na konkrétní situaci a scénáři použití. Jestliže víme, jaký máme HW a SW, a víme, k čemu toto zařízení má sloužit, pak teprve můžeme maximálně optimalizovat daný start a poskytnou tak funkční systém s efektivním a rychlým startem.

2.1 Optimalizace zavaděče

Proces zavedení jádra operačního systému se dá konfigurovat. Pro GRUB k tomu slouží soubor /etc/grub. Z důvodu kompatibility se staršími disky doporučuje použití "krátkého" čekání (asi 3s), než se disk připraví k použití [3]. Pokud bychom nepočkali, systém by registroval chybové stavy a start by trval déle, nebo by byl dokonce neúspěšný. U moderních disků, zejména u SSD, je tento čekací interval zbytečný. Nastavení tohoto parametru povede jednoznačně k úspoře startovacího času. GRUB taktéž obsahuje nastavení GRUB_CMDLINE, kde je možné nastavit informační výpis jako tichý a nacházejí se zde i nepotřebné parametry, které je možné odebrat. Vhodné je taktéž odebrání startovacího loga (SPLASH SCREEN) a podobných grafických prvků. Nejen že se zde praktikuje umělé pozdržení startu z marketingových důvodů, ale taktéž je start ovlivněn zbytečnými datovými přenosy.

Taktéž GRUB nabízí možnost optimalizace pro souborový systém EXT4 . Proces startu OS může být rychlejší díky využití defragmentace a nástroje Ext4 – Reducing Access Times, zkráceně e4rat. Pro systém souborů EXT3 existuje možnost povolného zpětného zápisu. Tato funkce ve většině systémů povolena není a musí se explicitně zapnout. Tím získáme opět časovou úsporu. U starších disků, které zpětný zápis nepodporují, bychom ale povolením této funkce obdržely opačný výsledek. Proto je tato funkce explicitně zakázán.

Zavaděč můžeme taktéž nakonfigurovat pomocí vrstvy Yocto. Tyto vrstvy totiž konfigurují systém na míru a mohou tak i ovlivnit zavaděč a jeho konfiguraci skrze recepty. Například jsem zjistil, že vrstva Yocto pro Raspberry Pi poskytuje tuto možnost a toho jsem později také využil.

2.2 Optimalizace jádra a modulů

Konfigurace jádra Linux může při startu OS taktéž přinést čas navíc. Jádro je komplexní a díky konfiguraci můžeme připravit systém na míru našim potřebám včetně optimalizace staru OS [2]. Jeden systém můžeme v různých konfiguracích provozovat optimalizovaně jak na serveru, tak i třeba na Raspberry Pi. Mezi první optimalizace se dá zařadit odstranění komprese initramfs. Taktéž můžeme redukovat velikost jádra výběrem sdílených knihoven, a to nejen výběrem alternativy ke knihovně libc, která existuje i ve velmi odlehčené verzi. Taktéž je možné změnit alternativní kompresi jádra ke Gzip, jako je například komprese LZO, která cílí na rychlou dekompresi. Ve snaze o co nejmenší rozměr jádra můžeme zakázat nepotřebné HW kousky, jako může být například SDHC nebo SATA kontrolér. Na závěr všech optimalizací je taktéž vhodné odebrat podporu ladění jádra při startu [1].

Zadání práce požaduje nejvýznamnější optimalizaci v této fázi startu, a to volbu modulů jádra. Díky tomu, že pracuji s moderním jádrem, které je modulární a obsahuje v sobě minimum funkcí, mohu svobodně přidávat a odebírat rozšiřující funkce v podobě modulů. V opačném případě by byly všechny ovladače součástí jádra. Díky modulům mohu kernel naučit bez kompilace nové funkce a mohu využít technologii Plug-and-play. Primárně jsou tedy použity pro funkce systému souborů a pro ovládaní přídavných zařízení. Pracují v nechráněném režimu. Jádro Linux je tedy monolit, který umí pracovat s moduly, a tak se dá dynamicky rozšiřovat. Výhodou této architektury je rychlost. Veškeré funkce jádra jsou implementovány do jednoho programu. Monolitické jádro pracuje v souvislém paměťovém prostoru a nemusí řešit přechody. Monolitické jádro, které umí pracovat s moduly, má také ale nevýhodu v podobě fragmentace paměťového prostoru. Volíme tedy jen ty moduly, které opravdu potřebujeme, a tak, aby šly odebrat. Nechceme je do jádra přímo přeložit [2].

Rozhodl jsem se tedy analyzovat moduly, které se nacházely v OS, který jsem optimalizoval. Pro výpis modulů nacházejících se v jádře jsem využil příkaz lsmod. Ten umí vypsat i závislosti mezi jednotlivými moduly. Taktéž jsem zjišťoval, jaké moduly je možné nalézt v adresáři /lib/modules. Následně jsem interaktivně detekoval chování při odebrání jednotlivých modulů a dopad na systém. Využil jsem k tomu příkaz

rmmod. V případě, že byl modul závislý na jiném, musel jsem nejprve eliminovat ten. Pokud jsem modul chtěl opět přidat, pak jsem využil dva příkazy insmode a modprobe. Příkaz insmode jsem použil pro základní práci se systémem. Musel jsem si ale sám hlídat závislosti tohoto modulu. Proto jsem vždy musel ručně zavést jednotlivé moduly, a to i ve správné posloupnosti. Díky modprobe jsem toto mohl dělat automatizovaně. Tabulka 1 shrnuje, jaký modul budu chtít z jádra kompletně odebrat. Každý modul má přiřazený krátký popis. Moduly, které jsem měl v plánu odebrat pomocí nástroje menuconfig, jsem vyznačil zeleně. Žlutě jsem vyznačil moduly se sporným odebráním. Červeně vyznačené moduly jsem se rozhodl neodebírat a nevěnovat jim dále pozornost.

Tabulka 1: Analýza modulů

Modul	Popis
rfcomm	Modul pro konfiguraci, správu a údržbu Bluetooth subsystému
bnep	Emulační vrstva nad BT pro ethernet (vyžadováno pro PAN)
hci_uart	Modul pro BT zařízení se sériovým portem
btbcm	BT podpora pro zařízení společnosti Broadcom
serdev	Základní podpora pro zařízení připojená skrze sériový port
bluetooth	Podpora pro BT subsystém
ecdh_generic	Generická implementace ECDH algoritmu
brcmfmac	Podpora pro bezdrátové LAN sítě a bezdrátový adaptér Broadcom FullMAX Chipset
brcmutil	Jedná se o Broadcom 802.11n Wireless LAN driver utilities
cfg80211	API WLAN konfigurace
rfkill	Modul kontrolující pomocí RF řadu Wi-Fi a Bluetooth karet
evdev	Event interface pro události zařízení
joydev	Rozhraní pro joystick
uio_pdrv_genirq	Ovladač pro I/O zařízení uživatelského prostoru s generickým vstupním kódem obsluhy
uio	Podpora I/O zařízení uživatelského prostoru (přerušení, paměť)
rpi_ft5406	Ovladač pro dotykovou obrazovku

fixed	Podpora pro fixní regulátor napětí
sch_fq_codel	Algoritmus pro plánování paketů
ipv6	Podpora pro IPv6 protokol

2.3 Optimalizace inicializačního skriptu a služeb

U inicializačního skriptu je vhodné minimalizovat nutnost volání systému. Ta přepínají mezi uživatelským režimem a režimem jádra. Tvoří tak časové prodlevy. Můžeme například při startu používat pouze vestavěné interprety příkazů [1]. Tím se redukuje forking, a tím pádem i počet volání systému. Proto je vhodné pro tyto skripty nepoužívat například Python. Eliminuje tak závislosti a máme menší systém souborů. Pomocí forkingu jsou implementovány linuxové roury, které proto není vhodné vyžívat právě v inicializačních skriptech. Proto je vhodné projít inicializační skripty a co nejvíce usilovat o optimální konstrukci skriptu pro start. U inicializace je vhodné minimálně odkládat data do oddílu SWAP. Pomoci zde může například extrémně rychlá paměť cache. K optimalizaci se dá využít i předlinkování knihoven. Taktéž je vhodné paralelizovat provádění jednotlivých úkonů inicializační fáze. Zde je nutné ale uvést předpoklad na procesor, který má více jader a vláken. V opačném případě lze očekávat horší výsledek [2].

Systémové služby, které jsou spouštěny inicializačním skriptem, respektive démonem Systemd u moderních distribucí, představují většinou největšího konzumenta času při startu OS. Optimalizace má zde prakticky podobu vypnutí nepotřebných služeb. Nesmí se ale zapomenout na skutečnost, že vypnutí služby, kterou systém nutně potřebuje, může vést k neúspěšnému startu či k nezdravému systému, který neposkytuje požadovanou funkcionalitu. Taktéž lze změnit pořadí startů jednotlivých služeb a tím eliminovat prodlevy, kdy jedna služba čeká, než naběhne jiná [1].

U svého systému jsem se tedy rozhodl vytvořit si tabulku, do které jsem si sepsal služby systému (viz Tabulka 2). Do tabulky jsem vždy uvedl službu a její podpis a barevné znázornění, zda tato služba poskytuje prostor pro optimalizaci. Nejčastější jsem uvažovat optimalizaci v podobě odebrání spuštění služby. Zeleně jsem vyznačil služby

s prostorem pro optimalizaci, červeně služby, které jsem se rozhodl neoptimalizovat a žlutě služby sporné a vhodné pro ověření optimalizace. Inicializační služby jsem získal díky příkazu systemd-analyzed. Taktéž mi s detekcí služeb pomohl příkaz systemctl a pro práci s logy journalctl. Tyto služby jsem musel zjišťovat pro první iteraci startu, tak i pro druhou. V obou případech se totiž spustily jiné kombinace služeb.

Tabulka 2: Analýza služeb

Služba	Popis
dev-mmcblk0p2.device	Start z diskového oddílu kary SD
hciuart.service	BT UART
systemd-hwdb-update.service	Správa databáze HW
systemd-fsck-root.service	Kontrola systému souborů rootfs
systemd-sysusers.service	Vytvoření a alokace uživatelů a skupin
systemd-journal-flush.service	Flush žurnálovacího démona
systemd-logind.service	Správa uživatelských přihlášení
ofono.service	Podpora pro mobilní zařízení
ldconfig.service	Odkazy a cache pro často použité sdílené knihovny
systemd-networkd.service	Síťový manažer
systemd-timesyncd.service	Síťová synchronizace lokálního času
systemd-resolved.service	Překlad hostname a adres IP
systemd-udev-trigger.service	Správa událostí zařízení
run-postinsts.service	Post inicializační přidělení
systemd-remount-fs.service	Připojení disků dle tabulky fstab
systemd-machine-id-commit.service	Zápis ID stroje do FS
systemd-random-seed.service	Uložení a načtení systémového náhodného semínka
systemd-journald.service	Sběr a ukládání logovacích data
systemd-journal-catalog-update.service	Aktualizace katalogů
tmp.mount	Připojení dočasného adresáře do FS

avahi-daemon.service	Implementace ZeroConf architektury
systemd-update-utmp.service + systemd-update-utmp-runle-vel.service	Zapisuje změny běhových úrovní SysV do utmp a wtmp adresářů Také provádí audit s logy
systemd-sysctl.service	Konfiguruje parametry jádra při startu
systemd-tmpfiles-setup.service	Vytvoření, smazání a vyčištění volatilních a dočas-ných souborů a složek
systemd-udevd.service	Správa událostí zařízení
kmod-static-nodes.service	Stará se o statické soubory reprezentující zařízení
bluetooth.service	Bluetooth
systemd-rfkill.service	RF kill switch
sys-kernel-config.mount	Konfigrace jádra
sys-kernel-debug.mount	Ladění jádra
systemd-tmpfiles-setup-dev.service	Podobně jako tmp předtím
var-volatile.mount	Připojení volatilního adresáře var
systemd-update-done.service	Implementace off-line aktualizace
dev-mqueue.mount	POSIX fronta zpráv FS
systemd-hostnamed.service	Démon pro kontrolu hostname z programů

2.4 Ostatní místa

Optimalizace sítě, do které je zařízení připojené, a správná konfigurace síťových nastavení rovněž vede k rychlejšímu startu OS. Vhodným přístupem je ale i minimalizace síťové komunikace. Mezi doporučení se řadí primární použití protokolu IPv4 a protokol IPv6 zakázat. Taktéž může chybný obsah souboru /etc/hosts, kdy na prvním řádku není záznam 127.0.0.1 localhost, vést k časovým prodlevám [4]. Režii síťové komunikace je možné snížit odebráním časových razítek a selektivní volbou paketu. V případě, že by zařízení využívalo starší přístup do sít, vytočení telefonní linky, selektivní volba paketu by znamenala zhoršení startovacího času. Zvážit se taktéž dá i zákaz určitých logování a ukládání metrik, ale to jen v případě, že o tyto cenné hodnoty není zájem.

Poslední optimalizací je iluze. Nejedná se již o přímé zefektivnění startu. Tato technika je založena na co nejrychlejším poskytnutí funkčního systému, nikoliv kompletně funkčního systému [5]. Spuštění tedy funguje tak, že uživateli je nabídnuta možnost pracovat se systéme v čase, kdy ještě nenastartovaly všechny služby. Zde může ale snadno dojít k hazardu. Například uživatel bude chtít poslat data po síti, ale síťový démon nebude ještě spuštěn. Data se neodešlou a uživatel nebude vědět, zda je problém v síti nebo v odeslání (aplikace, démon). Stav OS taktéž můžeme uložit do nevolatilní paměti, ze které si stav při příštím startu rychle načteme. Toho například využívají virtuální stroje. Úzkým místem zde může být nevolatilní uložiště a datový přenos do volatilní RAM paměti. U mobilních zařízení se často praktikuje uspání, zhasnutí displeje, přechod do nečinnosti. Nemusíme tedy systém přímo vypnout. Zde získáme extrémně rychlé naběhnutí systému. Ve skutečnosti jsme ale zařízení nevypnuli. Možností je taktéž vytvořit si aplikaci napsanou v jazyce C, která se při inicializační fázi spustí jako jediná a postará se o své závislosti. Jedná se vlastě o kompletní nahrazení inicializačního démona [6].

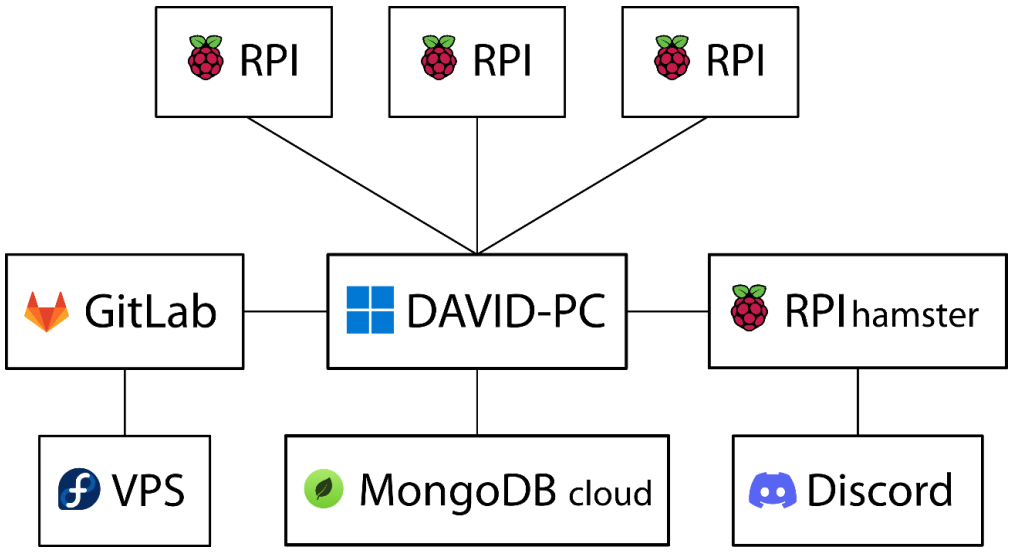
3 Zrychlení prakticky

V této kapitole jsem popsal způsob, jak jsem prakticky optimalizoval operační systém Linux dle navržených postupů. Taktéž jsem se rozhodl uvést postup přípravných prací, které mě dovedly k sestavení ekosystému.

3.1 Přípravné práce

Úvodem jsem začal tedy s přípravnými pracemi. Rozhodl jsme přistoupit k zadání komplexně. Zamyslel jsem se nad všemi úkony, které optimalizace zahrnuje. Uvědomil jsem si, že ruční zanášení naměřených dat do tabulky generované tabulkovým procesorem je zbytečně pracné. Čas, který bych strávil ručním zadáváním, jsem chtěl využít pro zkvalitnění jednotlivých optimalizací. Taktéž bych manuálním přepisem dat mohl zanášet nekorektní hodnoty do celkového souboru dat. Obdobné úvahy mě taktéž napadly pro sestavování a nasazování obrazu operačního systému.

Rozhodl jsem se tedy navrhnout a sestavit komplexní ekosystém. Jeho cíl byl jednoznačný. Ulehčit a zrychlit práci, automatizovat a zkvalitnit měření a následnou analýzu získaných dat. Obrázek 3 znázorňuje topologii ekosystému.



Obrázek 3: Topologie ekosystému

Položka DAVID-PC reprezentuje můj domácí stolní počítač. Protože můj nativní operační systém je Windows 10 Pro, nemohl jsem pracovat na optimalizaci operačního systému Linux s potřebnými nástroji. Z tohoto důvodu jsem se rozhodl zprovoznit si

technologii Hyper-V, která zajišťuje virtualizaci, a mohl jsem tak jednoduše používat virtuální linuxový stroj ve Windows. Jeho konfiguraci naleznete v repozitáři práce. Konkrétně jsem zvolil operační systém s názvem Fedora 35 Server a jedná se pouze o mou preferenci. Experimenty s jiným operačním systémem jsem nedělal, protože to nebylo cílem práce. Nicméně celý ekosystém jsem do jisté míry navrhl tak, aby byl nezávislý na zvoleném operačním systému. Později jsem zjistil, že je lepší využít navíc ještě kontejnerizaci pro vytvoření čistého vývojového prostředí, které je přenositelné a jednoznačně definované. Na tomto stroji za podpory nástroje Docker interaktivně vyvíjím v projektu Yocto. Taktéž se zde provádí automatizované sestavení OS. Nasadil jsem zde také kontejnerizovanou frontend a backend aplikaci pro datovou analýzu. Aplikace backend aplikace komunikuje s externí databází v cloudu MongoDB.

Ekosystém je taktéž napojen na GitLab. Ten poskytuje repozitář pro soubory ekosystému. Tímto rozhodnutím jsem také umožnil použití nástroje GitLab CI/CD. To znamenalo, že jsem zvolil platformu pro automatizované sestavování a nasazovaní OS. Taktéž mohu automatizovaně generovat, transformovat a publikovat dokumentaci na externí VPS.

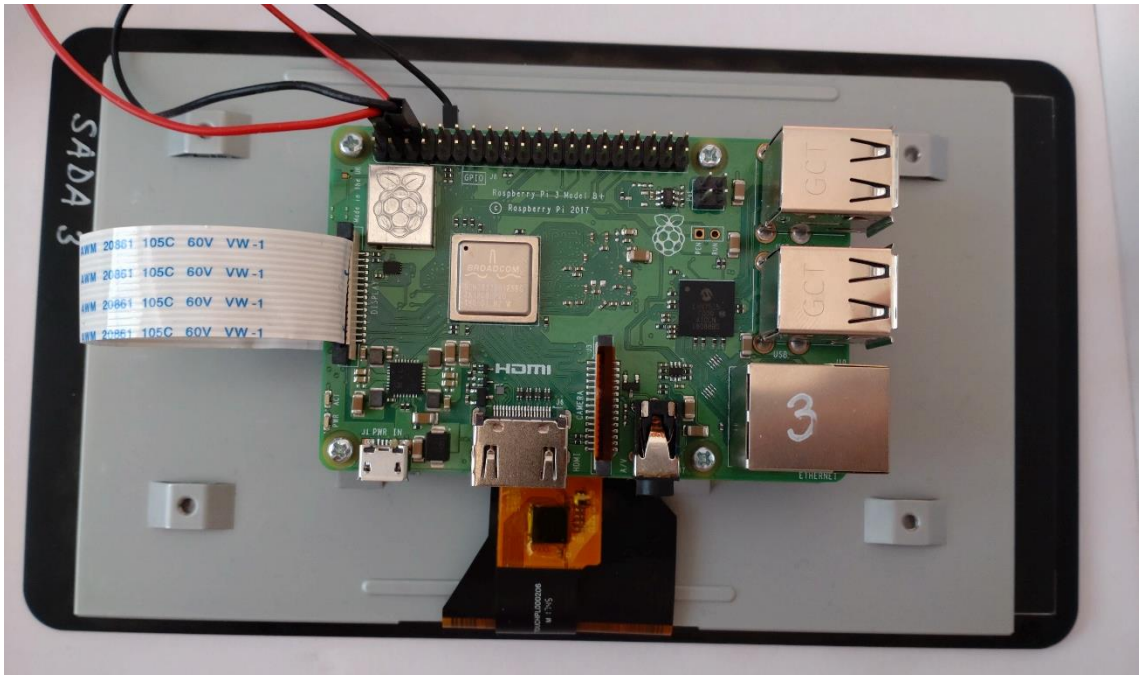
Pro nasazení obrazu OS jsem se rozhodl využít Raspberry Pi 4 opět s OS Fedora 35 Server. Toto zařízení jsem pojmenoval hamster. Pomocí SCP přenáším obraz z virtuálního stroje do stroje hamster. Zde jsem opět sepsal skript v jazyce Python, který využívá nástroj dd a může tak nasadit obraz na kartu SD. Skript je taktéž napojen na Discord. Jakmile dojde k nasazení, dostanu notifikaci, že mohu kartu použít. Z této karty SD následně startují koncová zařízení.

Na koncových zařízeních v obrázku označených jako RPI sbírám data pomocí skriptu napsaném v jazyce Bash. Tento skript získá data startu a odešle je na backend server.

3.1.1 Volba HW koncového zařízení

Pro začátek jsem se rozhodl zvolit HW platformu koncového zařízení. Zadání vyžadovalo ověření optimalizace na reálném zařízení reprezentovaném jako jednodeskový počítač s architekturou ARM. K dispozici jsem měl dvě zařízení, a to SAMA5D3 Xplained

a Raspberry Pi 3 Model B+. Raspberry má celkově širší komunitu koncových uživatelů [7]. Také mi vedoucí práce poskytla doplňkový 7″ dotykový displej. Proto jsem se rozhodl preferovat zařízení RPI.



Obrázek 4: Raspberry Pi s displejem

Raspberry Pi 3 Mode B+ stejně jako jiné zařízení RPI podporuje mnoho periférií. Primárně jsem využil možnost připojení externího displeje, který mi poskytla vedoucí práce. Pracoval jsem pouze s konzolí a možnost dotykové obrazovky tohoto displeje jsem nevyužil. Pokud bych ale pracoval se systéme s GUI, potencionál operační systém by byl mnohem větší a zavedení by trvalo déle. Pro ovládání jsem tedy primárně použil klávesnici připojenou skrze poskytované USB. Alternativou by bylo použití rozhraní Bluetooth. Zde jsem ale předpokládal, že Bluetooth nebude možné využívat po optimalizaci systému. Obdobně tomu tak bylo i u síťového rozhraní. Zjistil jsem, že zařízení RPI, se kterým jsem pracoval, mělo jak rozhraní Ethernet, tak i možnost bezdrátového připojení (Wi-Fi). Opět jsem upřednostnil fyzické médium. Důvod byl stejný jako u klávesnice. Navíc konfigurace připojení k mé domácí síti skrze bezdrátové připojení vyžaduje více konfigurace.

Pro zavedení operačního systému jsem se rozhodl využít preferovanou možnost, a to zavedení z karty SD [8]. Ekosystém tedy pracoval s touto skutečností a obrazy OS

sestavoval a nahrával na karty SD. Existují i alternativní možnosti zavedení OS pro toto zařízení. Já jsem se věnoval pouze variantě s kartou SD.

3.1.2 Organizace projektu

Celý projekt jsem potřeboval vhodnou formou organizovat. Cílil jsem na znovu použitelnost, přehlednost a flexibilitu. Využil jsem k tomu verzovací nástroj Git. Konkrétně jsem využil GitLab a díky tomu mohu mít projekt ve veřejném repozitáři a dávám tak možnost jak sobě, tak ale i ostatním můj ekosystém s optimalizacemi využít.

Alternativou nástroje GitLab by například mohl být GitHub, který ale nemá tolik otevřených možností. Nicméně pokud bych se v budoucnu přeci jen rozhodl migrovat z prostředí GitLab do jiného prostředí, bude to možné.

Navíc mi také GitLab poskytl možnosti jako dokumentační platformu GitLab Wiki a nástroj GitLab CI/CD. GitLab také poskytuje řadu další nástrojů, které bych mohl v budoucnu využít. Také se oproti konkurenci dá využívat i mimo v základu poskytovaný cloud [9].

Repozitář jsem se snažil logicky strukturovat tak, aby adresář vždy napovídal, co je jeho obsahem. VM obsahuje konfiguraci virtuálního stroje pro Hyper-V. Umožnil jsem tak snadnou distribuci nastavení virtuálního stroje Poseidon. Adresáře cache a home jsou určeny pro perzistentní data, která se vkládají do Docker kontejnerů při interaktivním vývoji i při automatizovaném sestavení. V configs se nacházejí konfigurační soubory jádra a recepty obsahující fragmenty nastavení jádra. Veškeré soubory týkající se kontejnerizace se nacházejí v adresáři docker. Chybí zde pouze konfigurační soubory typu .env, který jsem se rozhodl nepřidat do repozitáře. Tuto skutečnost jsem se rozhodl alespoň zdokumentovat a v dokumentaci popsat, jak si vytvořit vlastní .env soubor. Stejná situace platí i pro SSH klíče. V adresáři docs se nacházejí dokumentační nástroje a texty dokumentace. Frontend a backend pro analýzu dat a prezentační rozcestník se nachází v servers. Do adresáře tools jsem umístil nástroje, skripty a moduly, které celkově pomáhají s nastavováním, instalováním, klonováním, sestavováním, nasazováním a podobně. Jsou to praktické nástroje pro automatizaci nejen často používaných funkcionalit. Ve workspace se nachází kompletní pracovní adresář pro projekt Yocto.

Pro možnost paralelizace, lepší přehlednost a pro řízení automatizace jsem si vyčlenil názvosloví pro jednotlivé větve verzovacího systému. Obrázek 5 ukazuje tuto strukturu větví verzovacího systému. Název main říká, že se jedná o výchozí větev. V této větvi se také nachází stabilní podoba práce. Označení feature/ používám v případě umístění nové funkcionality do repozitáře. Pro demonstraci optimalizace přidávám ještě identifikátor optim a číslo optimalizace. Z této optimalizační větve dále vycházejí větev dokumentační s označením docs/ a větev pro konkrétní sestavení OS. Můžu tak paralelně dokumentovat, psát optimalizace, vyvíjet webovou aplikaci sestavovat několik OS a podobě.

```
main
|
|--> feature/optim-2
          ||
          ||--> docs/optim-2
          |---> feature/optim-2-yocto-build-work/rpi-fast-boot
```

Obrázek 5: Ukázková struktura větví

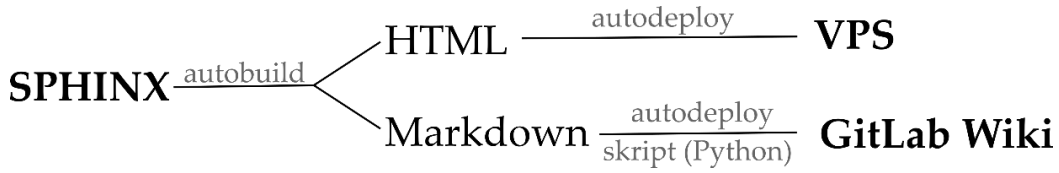
3.1.3 Dokumentace

Protože tato práce bude dále využita pro projekt, kterého se TUL účastní jako řešitel, rozhodl jsem se ekosystém a optimalizace zdokumentovat. Hlavním účelem tedy bylo, abych dal někomu návod, jak si může můj ekosystém zprovoznit a jak může sestavit jednotlivé optimalizace.

Obrázek 6: Ukázka dokumentace ve formátu HTML

Pro generování dokumentace jsme se rozhodl využít nástroj Sphinx. Byl pro mě uživatelsky velmi přívětivý a rychle jsem dosáhl potřebného výsledku. Texty jsem zapisoval ve formě reStructuredText. Jedná se odlehčený značkovací jazyk, který je se Sphinx spjatý [10]. Alternativou by mohlo být využití XML s formátem DocBook. Dokumentaci by pak měla být lépe udržitelná.

Publikovat jsem se rozhodl ve formátech HTML a Markdown. HTML jsem nastyloval pomocí Read the Docs. Inspirací pro mě zde byla řada dokumentací k OS Linux a k jazyku Python. Markdown formát jsem se rozhodl využít pro GitLab Wiki. Po vygenerování základního formátu bylo třeba soubor transformovat na formát pro GitLab Wiki. Některé časti se nevygenerovaly, či se vygenerovaly, ale vizuálně nesplňovaly požadovanou podobu. Například postranní navigace kompletně chyběla, odkazy byly chybné a některé elementy se vykreslily chybně. Tento problém jsem vyřešil transformačním skriptem napsaným v jazyce Python.



Obrázek 7: Schéma publikace

Abych snížil časovou náročnost procesu psaní dokumentace s následnou publikací, rozhodl jsem se opět o co nejvyšší míru automatizace. Využil jsem napojení na GitLab CI/CD. Ve finále mi tedy stačilo provést změny v textech dokumentace, tyto změny odeslat do repozitáře a GitLab CI/CD se mi postaral od generování v obou formátech, o transformaci Markdown souborů a o následné nasazení do staging prostředí, do GitLab Wiki a na produkčního VPS.

3.1.4 Vývojové prostředí

Pro tvorbu jednotlivých optimalizací jsem si připravil prostředí pro vývoj. Požadavky na toto prostředí jsem převzal ze zadání a z vlastních zkušeností při vývoji software. Bylo třeba, abych použil takové nástroje, které by mi umožnily pracovat přehledně a strukturovaně. Tak jsem vyřešil udržitelnost optimalizací a získal jsem možnost v budoucnu snadno navázat na již hotovou práci. Vývoj také musel probíhat v čistém

prostředí. Pokud by vývojové prostředí bylo kontaminované, objevily by se problémy při sestavování či nasazování operačního systému. Taktéž by se ale mohla zanést nechtěná úprava operačního systému, která by se neodhalila při těchto fázích vývoje. Mohlo by tak tedy dojít k závažnému ovlivnění dané optimalizace a tím pádem i výsledných naměřených dat, která byla klíčová pro vyhodnocení dané optimalizace.

Abych se vyhnul kontaminaci prostředí, využil jsem open source projekt pro automatizaci v podobě kontejnerizace Docker. Díky zabalení vývojového prostředí do kontejneru používám čisté systémové prostředí, které vždy vychází ze stejného stavu. Tento kontejner je taktéž ale i snadno přenositelný. Mohu tak nejen počet prostředí na jednom zařízení paralelizovat, ale i distribuovat na jiné stroje. Svou práci jsem tak připravil na týmový vývoj, na rychlý vývoj a na automatizaci sestavování a nasazování včetně testování.

Problematiku udržitelnosti kódu a dodržení pravidel správné struktury kódu jsem řešil skrze projekt Yocto a verzovací nástroj Git. Úpravy vedoucí k optimalizacím jsem se rozhodl uchovávat v receptech. Do receptů jsem taktéž umístil různé užitečné nástroje jako například konfigurační skript nastavující tyto optimalizace. Dalším příkladem může být recept, který do systému zahrne skript, který sbírá a odesílá naměřená data startu určená k pozdější analýze startu operačního systému. Tyto recepty následně shlukuji do vrstev, které následně poskytuji jednotlivým sestavením operačních systémů. Taktéž díky projektu Yocto, respektive díky nástroji devtool z projektu Yocto, mohu flexibilně, přehledně a udržitelně měnit konfiguraci jádra skrze nástroj menuconfig.

Výsledné soubory pak umisťuji do veřejného repozitáře poskytnutého platformou pro vývoj a provoz GitLab. GitLab využívá verzovací nástroj Git a veškeré změny tak mohu přehledně sledovat a lépe tak řídit vývoj či případné reverzní změny.

3.1.5 Projekt Yocto

Již zmíněný projekt Yocto používám pro optimalizování operačního systému Linux. Hlavním problém pro mě byla zprvu komplexita projektu. Musel jsem se seznámit s konkrétními postupy, které jsem potřeboval pro optimalizaci. V projektu Yocto je konfigurace rozdělena do tzv. vrstev, jež na sebe navazují. Díky tomu je možné snadno

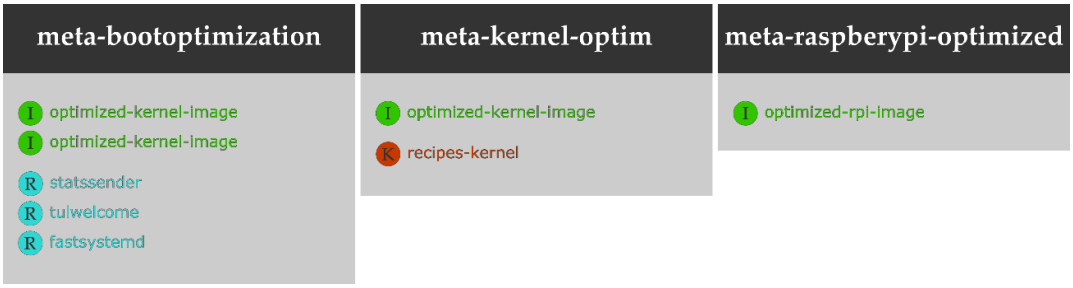
přidávat vlastní nastavení k již existující konfiguraci [11]. Já jsem pro bakalářskou práci vytvořil samostatné vrstvy definující jednotlivé optimalizace.

Pro tvorbu vlastního linuxového operačního systému využívám díky projektu Yocto referenční distribuci Poky. Tato distribuce je připravena a optimalizována pro vývoj vlastního operačního systému. Mám tak k dispozici základní jádro operačního systému se základními konfiguracemi [12]. Na základě této distribuce provádím napojení na rozšiřující vrstvy. Ty tuto distribuci upravují a připravují na použití na konkrétním HW.

Při práci jsem využil skript oe-init-build-env, který mi vytvořil zvláštní adresář pro práci. Současně mi korektně nastavil systémové prostředí. Tento skript mi taktéž umožnil korektně navázat na již existující pracovní prostředí. V pracovním prostředí vytvořeným skrze výše uvedený skript se nacházejí dočasné soubory, které se generují při sestavování operačních systémů. Pro mě zde byly ale klíčové dva konfigurační soubory. První bblayers.conf v sobě obsahuje informace o vrstvách, které mají být nástrojem BitBake využity pro konfiguraci v daném pracovním prostředí. Flexibilně tak mohu vybírat vrstvy, které mi přináší nové možnosti konfigurace. Druhým souborem je local.conf. Jedná se konfiguraci lokálního pracovního prostředí. Definuje se zde řada položek. Tyto položky ovlivňují výsledek sestavení operačních systémů skrze nastavení výchozí distribuce a zahrnutých vrstev.

Takzvané vrstvy v Yocto představují shluk nastavení, která lze flexibilně přenášet mezi pracovními adresáři projektu Yocto. Jedná se o metadata, která definují, jak cílový operační systém sestavit. Vrstvy jsou základem pro takzvaný model vrstev projektu Yocto. Tento model zajišťuje možnosti jako přizpůsobení systému, sdílení mezi vývojáři či znovupoužití dané vrstvy. Vrstvy zajišťují logické oddělení nastavení a informací pro pracovní prostředí. Vrstvy jsem tedy použil například pro udržení jednotlivých optimalizací tak, aby je bylo možné použít v různých sestaveních operačního systému. Mohl jsem snadno měnit kombinace optimalizací přehledně použitím dané vrstvy. Jiné vrstvy, které jsem využil, naopak přinesly možnost ovládat potřebný HW. Tyto vrstvy je možné naklonovat z externích repozitářů a často je nabízejí samotní výrobci cílového HW. Z toho taktéž vyplývá, že jsem získal možnost své změny udržovat i zvlášť mino

hlavní repozitář. Tuto možnost jsem ale záměrně nevyužil a všechny vrstvy uchovávám v jednom repozitáři [13]. Pro tvorbu vrstvy jsem využil nástroj bitbake-layers.



Obrázek 8: Struktura vrstev a receptů

Uvnitř vrstev se nacházejí recepty, které v sobě nesou samotné úlohy, které upravují konfiguraci výsledného systému. Úloha do_build je základem každého receptu a závisí na ostatních úlohách potřebných k sestavení receptu. do_compile kompiluje zdrojový kód a takto existuje více úloh. Většinou uvedených zkratkou do_. Využil jsem oficiální dokumentaci projektu Yocto proto, abych se zorientoval v jednotlivých úlohách. Celkově se recepty dělí na klasické, které například tvoří souborový systém. Dále pak recepty pro jádro OS, díky kterým jsem mohl využít fragmenty z devtool a nastavit tak optimálně jádro. Třetí skupinou jsou recepty, které svým obsahem tvoří obrazy operačních systémů. Do těchto posledních receptů jsem vybíral, jaké všechny recepty se mají použít. Například když jsem chtěl používat editor Vim, pak jsem musel dopsat recept přidávající Vim.

Nástroj devtool poskytuje možnost sestavit, otestovat a zabalit SW. Výsledkem je tedy snadná integrace do obrazu operačního systému. Díky devtool jsem tedy mohl provést změny jádra operačního systému pomocí nástroje menuconfig a tyto provedené změny zachovat jak speciální recept. Obsahem tohoto receptu byly vygenerované fragmenty, které v sobě nesly jednotlivé úpravy provedené v menuconfig. Zprvu jsem vždy pracoval ve speciálním pracovním prostotu, a jakmile jsem byl rozhodnut pro aplikaci změn, převedl jsem tyto změny na výsledný obraz operačního systému. Takto jsem tedy vyřešil problematiku podsunutí konfiguračního souboru jádra správnou cestou. Alternativní cestou by pak bylo dodání vlastního nakonfigurovaného jádra, či celý vygenerovaný konfigurační soubor jádra. Z mého pohledu by to bylo ale dlouhodobě neudržitelné, a to je zásadní nedostatek.

Veškeré optimalizace jsem se rozhodl reprezentovat následující strukturou. Do adresáře workspace jsem umístil tři sestavení operačních systémů. Sestavení rpi-fast-boot jsem zaměřil na práci se službami a s výběrem z různých optimalizací. Sestavení rpi-kernel-optim jsem určil k optimalizaci jádra. Sestavení věnující se optimalizaci vrstvy pro RPI jsem pojmenoval rpi-layer-optim. Dále se v adresáři sources nacházejí jednotlivé vrstvy. Jsou zde jak mnou vytvořené, tak i ty, které jsem potřeboval využít z externích zdrojů. Mezi naklonované externí vrstvy jsem zařadil poky. Jedná se o vrstvu sreferenční distribucí OS Linux. Dále využívám meta-raspberrypi. Tato vrstva v sobě nese zejména podporu pro HW, respektive pro RPI. Poslední externí vrstvou je meta-openembedded.

Do meta-bootoptimization jsem umístil recept fasystemd. Tento recept jsem napsal tak, aby nakopíroval do cílového souborového systému skript s konfiguračním souborem. Cílem skriptu je nastavení systémových služeb dle výběru z konfiguračního soboru. Zavedl jsem tři možné hodnoty pro každou službu. Hodnotu nula používám pro informaci, že se s danou službou nemají provádět další úpravy. Hodnotu 1 jsem rezervoval pro zakázání této služby a hodnotu "R" používám pro reverzi provedené optimalizace. Součástí jsou taktéž recepty na samotné sestavení obrazů operačního systému, které jsem pojmenoval optimized-image.bb a optimized-systemd-image.bb. Hlavním rozdílem jsou mnou použité kombinace optimalizací, které bych zmínil dále v textu. Pro účel prezentace jsem si taktéž vytvořil uvítací obrazovku pro konzoli, kterou reprezentuji v podobě loga TUL ve formátu ASCII art. Recept tulwelcome jsem napsal tak, aby mi soubor s logem automaticky zanesl do souborového systému cílového zařízení. Komplexnější obsah, který jsem umístil do receptu statsender, je klíčový pro analýzu startu. Základním souborem je skript na získání a odeslání dat. Tento skript je možné opět konfigurovat skrze konfigurační soubor. Konkrétně se dá nastavit reakce na chybějící síťovou službu, požadavek na automatický restart po odeslání naměřených dat, poznámka měření a adresa serveru, se kterým se má komunikovat. Automatické nastavení DHCP pro síťové rozhraní eth0 jsem realizoval v podobě nakopírování soboru wired.network do adresáře s konfigurací sítě uvnitř adresáře /etc/systemd/network. To provádí taktéž recept statssender. Poslední úpravou souborového systému je pak přidání souboru cicdid.txt, který obsahuje automaticky vygenerovaný identifikátor

z CICD, a skript na odeslání dat, také automaticky odesílá údaje spjaté s konkrétním buildem.

Dále jsem ve vrstvě meta-kernel-optimization vytvořil recept optimized-kernel-image.bb, který popisuje sestavení obrazu operačního systému, který využívá automaticky optimalizaci jádra operačního systému. Tato optimalizace je umístěna recipes-kernel/linux v podobě soboru bbappend. Tento soubor automaticky provede změny uložené jako fragmenty. Tuto součást jsem automaticky vygeneroval za pomoci již zmíněných nástrojů devtool a menuconfig.

Ve vrstvě recipes-rpi-optimized jsem vytvořil pouze jeden recept, a to na sestavení obrazu operačního systému, který má optimalizovanou vrstvu pro RPI. Optimalizace jsou zde sepsány přímo v receptu.

3.1.6 Sestavení a nasazení OS

Jak jsem již uváděl, sestavení požadovaného obrazu operačního systém probíhalo vždy na mém stolním počítači uvnitř virtuálního stroje s operačním systémem Fedora. Abych se vyhnul kontaminaci pracovního prostředí, pracoval jsem uvnitř kontejneru běžícího ve službě Docker. Zprvu jsem prováděl proces sestavení manuálně, neboť nebyl nijak komplikovaný. Používal jsem nástroj bitbake, který mi pomohl se sestavením požadovaného obrazu. Nicméně s narůstajícím počtem sestavení jsem se rozhodl často spouštěný proces ulehčit sepsáním skriptu v jazyce Bash. Vznikl tak tedy skript s názvem run-build.sh a jeho hlavním úkolem bylo pouze sestavení zadaného obrazu OS. Nicméně zde jsem již počítal s budoucím napojením na DevOps nástroje, které mi později zajistily velkou časovou úsporu a pomohly mi s koncentrací na danou problematiku. Skript jsem tedy doplnil o zabezpečené připojení skrze SSH s podporou SSH klíčů na server, kde probíhal vývoj a sestavení, respektive na OS Fedora v kontejneru v mém PC. Toto spojení jsem využil k tomu, abych spustil kontejner pro práci s projektem Yocto. Uvnitř pracovního kontejneru jsem následně spustil název skriptu, který jsem přebral jako vstupní parametr skriptu run_build.sh. Cílem přebraných skriptů je sestavit vždy jeden konkrétní obraz operačního systému. Taktéž bylo nutné skript run-build.sh doplnit o zachytávání chyb z SSH a o výpis časových značek pro analýzu doby sestavení. Taktéž jsem se rozhodl metadata jako název sestavení, název obrazu, časy a podobně ukládat do

databáze MongoDB tak, abych na tato data mohl napojit jednotlivé straty daného sestavení. Proto jsem skript napojil ještě na skript image_build_report.py napsaný ve skriptovacím jazyce Python. Výsledkem fáze sestavení je tedy výsledný obraz operačního systému s metadaty uloženými v databázi podporující flexibilní strukturu dat a optimalizovanou pro velké množství dat.

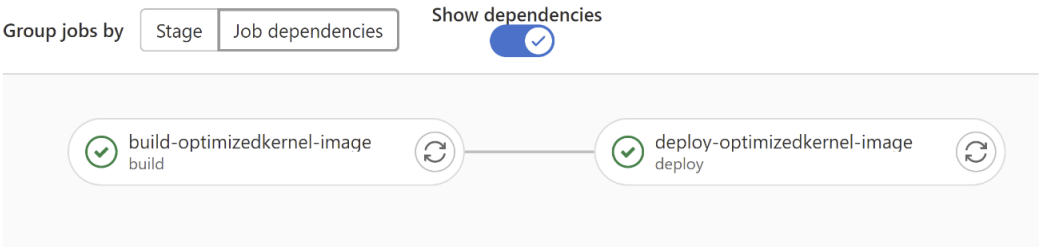
Na základě tohoto sestavení jsem mohl pokračovat a nasadit obraz operačního systému tak, aby jej bylo možné použít na cílovém HW. RPI podporuje více typů nasazení [8]. Já se rozhodl pro zápis na kartu SD. Důvodem byla nízká komplexnost a komunita okolo RPI, tento postup vřele doporučuje [14]. Pro nasazení jsem nejprve používal skript napsaný v jazyce PowerShell. Navrhl jsem jej tak, abych mohl nasazovat přímo ze stroje DAVID-PC a aby jednotlivé parametry nasazení mohly být předány jako vstupní parametry, či z interaktivní konzole. Skript se připojil na virtuální stroj s OS Fedora a stáhl požadovaný soubor, který se má nasadit. K tomu jsem využil nástroj SCP. Data jsou tak přenášena bezpečně a automatizovaně. Po přenesení jsem využil nástroj dd, který zapíše přenesený obraz na cílové médium, kterým byla vždy karta SD. Zde bych ale časem narazil na omezení mého stroje s Windows. Proto jsem vytvořil Bash skript s názvem run-deploy.sh, který se skrze SSH připojí na můj domácí stroj RPI 4, na kterém jsem prováděl sestavení. Na tomto stroji jsem si zprovoznil OS Fedora a dodal jsem veřejný SSH klíč pro možnost připojení. Taktéž jsem připravil redukci z USB na kartu SD. Skrze SSH je na tomto stroji spouštěn skript, který provede nasazení skrze zápis pomocí nástroje dd. Skript dále zjistí metadata nasazení a odešle je do databáze. Výsledkem je tedy sestavený obrazu operačního systému, který je možné volitelně nasadit. Protože proces sestavení a nasazení může trvat i více jak 2 hodiny, rozhodl jsme se napojit nasazení na komunikační platformu Discord skrze webhook. V praxi jsem tedy po nasazení okamžitě obdržel upozornění na hotové nasazení.

Takto připravené nástroje se mi samy nabízely k využití automatizace. Chtěl jsem tedy pouze provést editaci nějakého souboru pro Yocto a tyto změny odeslat od repozitáře. Automatizace mi měla zajistit přijetí změn, sestavení obrazu OS a jeho následné nasazení. Ve finále mi tedy měla pouze přijít notifikace o úspěšném nasazení automaticky sestaveného obrazu operačního systému. Toho cíle jsem dosáhl, a to díky platformě GitLab. GitLab nabízí možnost využít automatizaci v podobě GitLab CI/CD pipelines.

Ušetřilo mi to práci tím způsobem, že jsem nemusel manuálně spouštět své obslužné skripty pro sestavení (CI) a pro nasazení (CD). GitLab mi oba skripty spustil sám v reakci na odeslání upraveného kódu. Proto, abych mohl využívat tuto službu, kterou GitLab nabízí, jsem musel nejprve nasadit kontejner v privilegovaném režimu s názvem GitLab Runner. Tento agent se stará o komunikaci s platformou GitLab a spouští požadované úkony. Rozhodl jsem se preferovat kontejnerizaci a úkony provádím v kontejnerech. Sestavení tedy probíhá v jednom kontejneru a nasazení v jiném. GitLab Runner má alternativu, a to v podobě veřejných strojů nabízených platformou GitLab. Zde jsem se ale potkal s omezeným využitím. Zdarma je totiž pouze určitý počet hodin a některá má sestavení trvala i více než 3 hodiny. Druhou ingrediencí k úspěchu pak byla nutnost připravit soubor .gitlab-ci.yml. Do tohoto souboru jsem definoval fáze sestavení a nasazení pro každou optimalizaci zvlášť. Nastavil jsem omezení, která říkají, že pokud by při sestavení došlo k problému, nasazení se neprovede. Volba obrazu operačního systému, kterou chci sestavit a nasadit, se odvíjí od toho, v jaké větvi verzovacího systému pracuji. Nesestavují se mi tak zbytečně obrazy OS, které nepotřebuji a zároveň nemusím řešit kolize při zápisu na stejné médium. Automatizace tedy postupně projde jednotlivé položky uvnitř souboru YML. Pokud položka splňuje pravidlo s větví, začne se sestavovat. Spustí se již zmíněný skript na sestavení a odeslání metadat. Do kontejneru mám napojený adresář s naklonovaným repozitářem, takže data ze sestavení jsou perzistentní. Následně se na tato data napojí další fáze, kterou je nasazení, které opět splňuje podmínku na požadovanou větev. Nasazení se připojí na můj domácí stroj RPI, na kterém se spustí nasazovací skript. Proto abych se mohl napojit na své domácí vývojové prostředí, jsem opět využil SSH, konkrétně jsem využil odlehčené kontejnery s OS Alpine, které se připojí přímo do vývojového prostředí a mohou pracovat na sestavování a na nasazování.

Celý proces lze interaktivně sledovat skrze platformu GitLab. Zároveň jsem mohl kontrolovat, zda vše proběhlo, jak má, či nikoliv, a pokud byla jedna z fází neúspěšná, tak jaký byl problém. GitLab totiž uchovává historii běhů jednotlivých fází. Nicméně jsem navíc napojil GitLab na komunikační platformu Discord, kam se mi již zasílala informace o úspěšném nasazení. Nově se mi tak odesílaly i informace o CI/CD. V praxi to pro mě znamenalo možnost optimalizovat či měřit straty a nemusel jsem kontrolovat,

zda už došlo k úspěšnému sestavení s následným nasazením. O stavu a průběhu mě vždy informoval Discord.



Obrázek 9: GitLab CI/CD a zobrazení úloh

3.1.7 Analýza dat startu OS

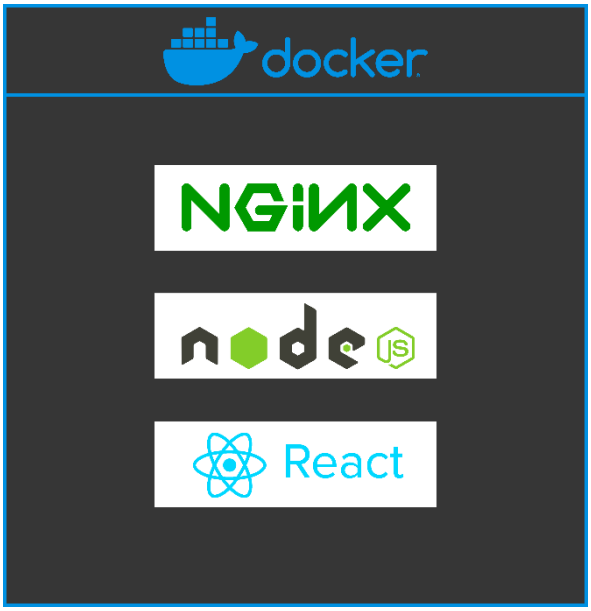
Data generovaná při startu OS byla pro mě zásadní a veškeré vyhodnocení optimalizací bylo na těchto datech založeno. Měření jednotlivých optimalizací bylo třeba řešit odděleně a následně v různých kombinacích. Aby bylo možné s daty pracovat rychle a přehledně, bylo taktéž potřeba vymyslet sparné uložiště a vhodnou formu prezentace těchto dat.

Samotný sběr dat jsem vyřešil pomocí skriptu, který pomocí nástroje systemd-analyze zjistil informace týkající se startu. Konkrétně jsem tak tedy získal totální čas startu OS, dále pak časy, kdy systém při startu využíval uživatelský prostor a prostor pro jádro. Taktéž jsem si získával latence při startu způsobené jednotlivými službami, které mi pomohly s identifikací problémových služeb. Skript jsem vytvořil tak, aby mi data převáděl na formát JSON. Připravenou datovou strukturu doplněnou o data ze sestavení a jiné podrobnosti jsem skrze nástroj curl odesílal na server, který se dále staral o zpracování a uložení těchto dat.

Server na zpracování, uložení, práci a na vizualizaci dat jsem taktéž vytvořil jako součást této práce. Pro snadné nasazení s vysokou mírou flexibility jsem využil opět kontejnerizaci a pomocí souboru docker-compose.yml jsem si vytvořil stack se dvěma kontejnery. Tyto kontejnery mají propsané porty dle potřeby tak, aby na ně bylo možné přistoupit skrze reverzní proxy Nginx. To mi v praxi umožnilo přistupovat skrze doménu přímo ke službě uvnitř kontejneru. Současně bylo třeba také oddělit testovací a

produkční verzi nasazení stacku, a proto jsem využil systémové proměnné nastavené souborem .env.

První kontejner obsahuje aplikaci označenou jako backend. Jedná se server, který jsem napsal v jazyce NodeJS. Pomocí .env souboru mu předávám potřebné konfigurace systémových proměnných tak, abych nemusel v repozitáři uchovávat citlivé údaje. Server komunikuje s externí databází MongoDB realizovanou jako externí služba uvnitř architektury cloud. Nicméně díky kontejnerizaci by neměl být v budoucnu problém použít vlastní hostovanou instanci této NoSQL databáze. Pomocí definice jednotlivých cest jsem si vytvořil jednoduché API, na které přistupuje výše uvedený skript, který posílá soubor JSON. Server tedy data přijme, provede jejich finální úpravu a uloží je skrze MongoDB driver do databáze. Jako součást API jsem taktéž vytvořil možnost získat data z databáze pro libovolné užití. Tuto funkci jsem primárně určil pro aplikaci na datovou analýzu a na vizualizaci těchto dat. Díky zvolené architektuře nebude v budoucnu problém přejít na jinou formu této aplikace například napsanou v jazyce Go. Klienti připojující se na toto API by neměly poznat žádný rozdíl.



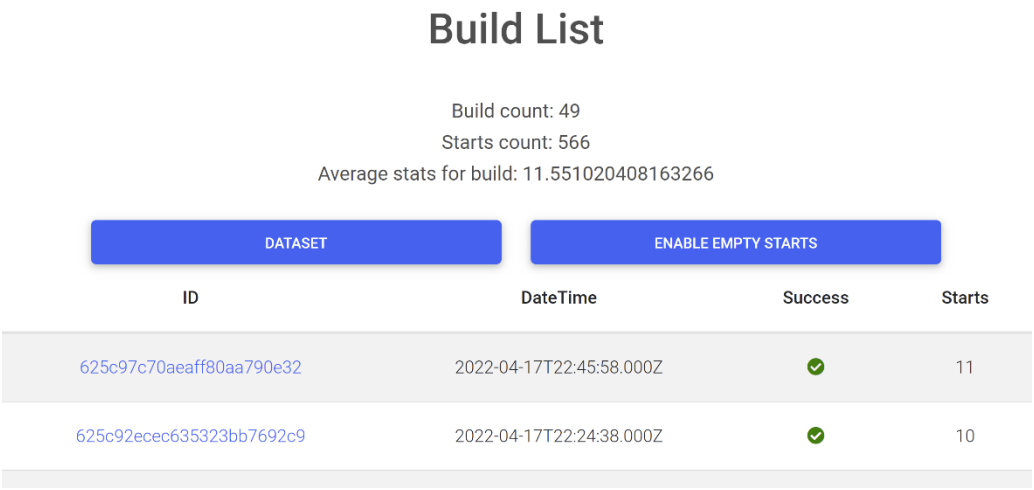
Obrázek 10: Schéma kontejnerů v Dockeru

Druhým kontejnerem je aplikace frontend. Zde jsem využil technologii React, která je určena právě primárně pro tvorbu aplikací označených jako frontend [15]. Zde bylo taktéž třeba, abych daný port propsal ven z kontejneru a mohl se tak připojit

z internetu skrze reverzní proxy. Taktéž zde využívám .env soubor, který ale musí obsahovat specifickou nomenklaturu, kterou vyžaduje React. Tohoto klienta jsem napsal tak, aby se připojil na server, zde si v případě potřeby vzal data opět ve formátu JSON a dále s těmito daty operoval. Komunikaci jsem realizoval jako asynchronní. Data se tedy načítají postupně. Zde jsem narazil na zpomalení, které způsobuje použití databáze v externím cloudu. Proto jsem usoudil, že by bylo lepší si zajistit lokální instanci MongoDB v kontejneru, tento kontejner napojit na stack a upravit server tak, aby komunikoval s touto instancí DB. Pro plynulost přechodů jsem využil routování a takzvané links. To mi pomohlo s úsporou načítaných dat a se sledováním historie. Taktéž jsem využil moderní přístup k napsání tohoto klienta za pomocí hooks. Dosáhl jsem tak lepší čitelnosti kódu. Pro vzhled jsem využil framework Material Design for Bootstrap v5 v edici pro React. Získal jsem tak řadu komponent, které jsem mohl využívat či upravit jejich definici dle potřeby a dále využívat. Pro snadnou práci s daty jsem taktéž využil open-source knihovnu APEXCHARTS, která mi poskytla opět komponenty pro React, které jsem mohl využít při datové analýze.

Jako index webu poskytovaného klientem React jsem vytvořil takzvaný Build List. Zde jsem vytvořil tabulku, která přejímá načtená data sestavených obrazů OS, která jsou seřazena podle toho, jak byla do DB vložena. Pro rychlou orientaci jsem se liché a sudé řádky rozhodl barevně oddělit a taktéž zvýraznit řádek, na který ukazuje kurzor myši. Do řádků jsem si zvolil vypsání I, které má taktéž sloužit i jako reference na porobnosti daného sestavení, konkrétně na analýzu jednotlivých startů daného sestavení. Obsahem řádku je vždy taktéž datum a čas sestavení obrazu OS, ikonka signalizující, zda bylo sestavení úspěšné, a počet startů daného sestavení OS. Uživateli jsem se v základní situaci rozhodl nevypisovat prázdné starty. V případě potřeby jsem připravil tlačítko ENABLE EMPTY STARTS, které vypíše řádky s nulovými starty. Jako součást sekce Build List jsem také vytvořil výpis součtu všech sestavení, výpis součtu všech startů a výpis průměrného počtu startů na jedno sestavení. Statistiku dynamicky aktualizuji dle preference výběru výpisu řádků s prázdnými starty. Taktéž mě napadlo, že by si uživatel chtěl stáhnout data, se kterými aplikace pracuje. Uživateli jsem tedy vytvořil možnost stáhnout si dva soubory JSON. OS build obsahuje data sestavení a nasazení a OS starts obsahuje jednotlivé starty. Alternativou by pro uživatele pak bylo přímo využít backend

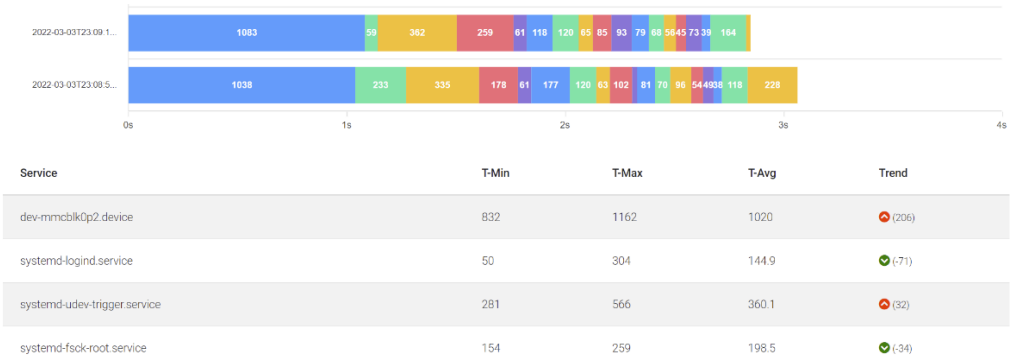
a tato data získat odtamtud, což je vhodné spíše pro automatizaci a odstranění potřeby klikání.



Obrázek 11: Build List

Pro samotnou analýzu jednotlivých startů daného sestavení jsem vytvořil oddělenou sekci, kdy úvodem vypisuji základní informace o datech získaných z CI/CD. Uživatel tak přesně ví, s jakým sestavením pracuje a pro jaké sestavení z Yocto a pro jaký obraz jsou vizualizovaná data určena. Na začátek sekce jsem se taktéž rozhodl zahrnout výpis celkového počtu startů, průměrné časy pro start OS, pro čas strávený v prostoru pro jádro, pro čas strávený v uživatelském prostoru a výběr nejrychlejšího a nejpomalejšího startu se základními údaji. Po výpisu obecných informacích jsem vytvořil tabulku s obsahem všech stratů. Vzhled jsem přejal od tabulky z komponenty Build List. Kromě ID a datumu s časem sestavení taktéž vypisuji popisek startu, který byl vložen při analýze na koncovém zařízení. Užitečné mi taktéž přišlo odfiltrovat si konkrétní start, a proto jsem do tabulky přidal ikonky na odebrání konkrétního startu. Prakticky to pro uživatele znamená, že se mu start odebere, a aplikaci jsem napsal tak, že automaticky překreslí data, grafy a jednotlivé výpisy aktualizuje. To znamená, že například průměrné hodnoty se automaticky přepočítají. To mi později velmi pomohlo s vyhodnocením jednotlivých optimalizací, a to nejen pro konkrétní typ restartu operačního systému. Taktéž jsem pro vyhodnocování potřeboval vykreslovat grafy, proto jsem použil komponenty z knihovny APEXCHARTS. Jako první jsem vytvořil sloupcový graf, který vizualizuje totální časy startů. Na osu x jsem umístil čas v sekundách a na osu y jsem zanesl datum a čas jednotlivých startů. Ve sloupci jsem se rozhodl taktéž vypsat celkový čas, a to

v milisekundách. K tomuto grafu jsem také umístil vizualizaci trendů pro celkovou dobu startu, pro dobu startu strávenou v prostoru pro jádro a pro dobu strávenou v uživatelském prostoru. Vizualizace je v podobě barevné ikonky se symbolem a s vypočteným skóre. Výpočet jsem definoval jako součet diferencí mezi jednotlivými starty. Ve druhém grafu, který jsem zahrnul, vizualizuji podobné informace, ale tentokrát rozdělené na časy strávené v uživatelském prostoru a v prostoru pro jádro OS. Díky knihovně mohu snadno odfiltrovat konkrétní prostor a vizualizovat si tak například pouze uživatelský prostor. Posledním grafem, který jsem vytvořil, je graf s časovými intervaly startů jednotlivých služeb. Zde je vždy sloupec pro daný start, který je barevně rozdělen na časové intervaly jednotlivých služeb. Opět uživateli nabízím možnost odebrání konkrétní služby. Jako poslední položku jsem vytvořil tabulku s obdobným vizuálním vzhledem, jako u předešlých tabulek, tentokrát ale s možností řazení dle sloupce. Tato komponenta mi totiž měla pomoci s analýzou časů jednotlivých služeb. Vizualizuji si tak pro každou službu řádek obsahující shrnující údaje ze všech startů pro danou službu. Konkrétně tedy minimální, maximální a průměrná doba. Nechybí také trend, kdy vizuální podobu a výpočet přebírám od trendu pro start.



Service	T-Min	T-Max	T-Avg	Trend
dev-mmcblk0p2.device	832	1162	1020	(206)
systemd-logind.service	50	304	144.9	(-71)
systemd-udev-trigger.service	281	566	360.1	(32)
systemd-fsck-root.service	154	259	198.5	(-34)

Obrázek 12: Analýza startů (GUI)

3.2 Konfigurace systémových služeb

Díky přípravě ekosystému a detekci start prodlužujících služeb jsem mohl postupně procházet a deaktivovat startovací časy jednotlivé služby. V této podkapitole popisuji, jak jsem optimalizoval start operačního systému v oblasti systémových služeb. Tabulka 3 obsahuje veškeré optimalizace s příslušným ID tak, aby si čtenář mohl dohledat a projít interaktivně naměřená data jednotlivých optimalizací a mohl si je vizualizovat skrze

aplikaci ekosystému. Řádky označené modře se vztahují k optimalizaci služeb, červeně jsou zvýrazněné řádky optimalizace vrstvy pro RPI, žlutě pak řádky optimalizace provedené volbou modulů a konfigurací jádra. Nepřiřazené záznamy jsem vyznačil zeleně. Řádek vždy obsahuje položku optimalizace a identifikační číslo sestavení, na kterém byla optimalizace měřena. Toto ID stačí vyhledat v aplikaci v Build Listu a čtenáři se tak vizualizují data, ze kterých jsem vycházel.

Tabulka 3: Reference optimalizací na data v aplikaci

Optimalizace	Sestavení
neoptimalizováno	6225de0e456c20f56ec7f5f3
hciuart	622b5ce2c781b7e20c6ab19e
systemd-fsck-root	622b8d2daa362374cb3eeefb
systemd-journal-flush	622b8e814e236bdaba548950
ofono	622bcd9ce33e9fe82e2416f3
systemd-networkd	622bd13eac22ac27a5447dc3
systemd-timesyncd	622c6217390622220a717740
systemd-resolved	622cc4791b3a24caabfa44c8
systemd-journald	622cd1856863da8daba275c8
tmp.mount	622cd8637e87fbf8f2198472
systemd-udevd	622cde0a331e457b6d7f1893
kmod-static-nodes	622d23cd4bcd29e6d2a99c12
bluetooth	622d29da5a08ba4eb22f35e0
systemd-rfkill	622e719495376c1764d6c8c5
dev-mqueue.mount	622e77858819ed057c9dedc3
sytsemd-hostnamed	622e7b7de5292f4eb46b853d
služby hromadně	622fc265b4689409b213584f
avahi	62307e7d2c5a22d0b76b35e7
zákaz splash screen	62436d7c236987933c772bc3
nastavení min. paměti GPU	6243791d9f3b373eb7cdc1f0
zakázání overscan	62438050b10aca45b486592d

boot delay	624383b261eb8dad814a60a3
přetaktování	624385ca349312e9532e2731
odebrání start. loga RPI	624388d455ccbda477589360
nast. vrstvy - vše	62439151cbd852f9d6ce16e2
nast. vrstvy - doporučení	62439315849d03f3b0050e68
LZO komprese	6255aaa9d647869d15d9ec84
bluetooth	625ad50bf13581260b12a155
veškeré wireless	625ae960c663bc5d9300ae94
IPv6	625afe959dcf7904a39ebcea
ladicí nástroje	625b46fa11eb791d334a5a6b
výsledná kombinace	625b4e68bc4a6ffd11cbc022

3.2.1 Měření

Při analyzování služeb a jejich optimalizacích bylo třeba, abych provedl několik typů měření. Konkrétně jsem musel změřit chování systému při prvním startu OS, restart vyvolaný přerušením napájení a restart vyvolaný příkazem reboot. Systém se vždy choval jinak, časový interval startu OS se taktéž lišil, a dokonce systém pracoval i s jinými službami. Abych tedy mohl vždy jednoznačně určit dopad optimalizace, musel jsem data naměřit čistě pro optimalizovanou verzi systému, ale i pro výchozí verzi systému. Abych mohl pracovat s větším vzorkem dat, musel jsem provést i více (re)startů daného typu. Z těchto hodnot mi šlo lépe udělat průměrné hodnoty, které dokázaly s lepší komparací systému. Obecně jsem tedy prováděl 1x první start OS, 3x až 10x restart vyvolaný přerušením napájení a 3x až 10x restart příkazem reboot. Nejčastěji se však jednalo o kombinaci 1x první start, 3x restart vyvolaný přerušením napájení a 5x restart vyvolaný příkazem reboot. U některých služeb je taktéž vidět navyšující či snižující se trend. Optimalizaci navyšujících se trendů jsem neřešil. Zaměřil jsem se na přímé hodnoty. Sestavil jsem tedy pomocí automatizace a ekosystému obraz OS, který jsem taktéž automatizovaně nasadil na kartu SD. Sytém souborů tohoto obrazu OS díky receptu obsahuje soubor cicdid.txt, ve kterém je umístěn identifikátor pro databázi. Data jsem tedy mohl přímo přiřadit k danému sestavení automaticky.

Taktéž jsem mohl díky receptu pracovat s analyzačním skriptem. Tento skript jsem pomocí konfiguračního nástroje upravoval tak, abych měření mohl přivádět přímo na míru aktuální systémové konfigurace, a to i v případě absence síťové služby. Zároveň jsem využil i druhý přídaný skript. Ten jsem mohl opět konfiguračním souborem upravit tak, aby mi skript vybral konkrétní službu či kombinaci služeb. Tento optimalizační skript jsem obvykle spouštěl hned po prvním startu. Dříve by ani spustit nešel. Z toho také plyne, že výběr služeb nikdy neoptimalizuje první strat OS. Postupoval jsem následovně.

Nejprve jsem změřil první start. Během práce jsem postupně došel k označování těchto měření. První strat jsem tedy popisoval jako firstboot. Naměřená data mi ukázala, že tato první iterace může být 3x delší než jiný klasický start. Nejdéle zde trvá start z oddílu karty SD, aktualizace databáze HW nebo například kontrola souboru systémů. Drtivou většinu času start tráví v uživatelském prostoru. Pro spoustu sestavení byl první start výrazně nejpomalejší.

Po prvním startu jsem odeslal s popiskem data a provedl jsem optimalizaci systémových služeb dle potřeby. Následně jsem proved restart, kdy jsem přerušil napájení zařízení. Konkrétně jsem vytáhl napájecí adaptér ze zásuvky a po chvíli jsem ho opět připojil. Zde bylo třeba, abych vždy počkal podobnou a dostatečně dlouhou dobu. Pokud jsem čekal výrazně kratší dobu, zařízení naběhlo zpravidla rychleji a měření optimalizace bylo zkreslené. Tento restart jsem se rozhodl pojmenovat hw reboot s poznámkou, co jsem optimalizoval. Zde jsem tedy už postupoval restart, odeslat data restart, odeslat dataatd. Čas oproti prvnímu startu výrazně poklesl a poměry prostoru jádra a uživatelského prostoru se pohubují v podobných číslech.

Po provedení všech HW restartů jsem se přesunul k měření restartů, jejichž příčinou byl příkaz reboot. Pro vyšší míru automatizace jsem ve skriptu využil možnost automatického volání příkazu reboot po odeslání dat. Těmto datům jsem přidělil obdobně označení sw reboot a název optimalizace. Globálně se jednalo o nejrychlejší typ startu. Dominantní zde byl uživatelský prostor.

3.2.2 Optimalizace jednotlivých služeb

První služba, na kterou jsem se zaměřil, byla hciuart. Jednalo se o velkého konzumenta času a odebráním této služby jsem získal úsporu cca 6,5 s. Stinnou stránkou optimalizace naopak bylo znefunkčnění služeb Bluetooth. Tuto služby mi stačilo pouze zakázat bez použití maskování.

Službu systemd-fsck-root se mi nepodařilo optimalizovat. Naopak jsem "optimalizací" mírně start prodloužil. O zrychlení jsem se pokusil úpravou tabulky /etc/fstab. Pokusil jsem se nastavit zákaz kontroly při startu.

Další v pořadí byla služba systemd-journal-flush, do které jsem nevkládal potencionál. Mnou provedená optimalizace spočívala v převodu dočasných logů do logovacích souborů, promazání logovacích souborů, nastavení omezeného uložiště pro logy. Závěrem jsem ještě službu zamaskoval. Výsledek byl úspěšný, ale časy jiných služeb se mírně navýšily. Výhodou taktéž byla menší velikost logovacích souborů. Nevýhodou pak nespolehlivost perzistentnosti logů.

Optimalizace služby ofono nepřinesla výrazné zrychlení, nicméně i 10 až 100ms se může v určitých případech hodit. Optimalizací jsem nepocítil žádné nedostatky při používaní zařízení. Službu stačilo zakázat.

Náročnější optimalizací, která mě původně vedla i k úpravě skriptu sbírajícího data, bylo zefektivnění startu systemd-networkd. Službu bylo nutné jak zakázat, tak i následně zamaskovat. Tím jsem se připravil o možnost síťové komunikace. To v praxi znamenalo, že mi žádný démon nepomohl s odesláním naměřených dat. Do skriptu jsem tedy integroval přepínač, který skriptu říká, zda se má automaticky odmaskovat a spustit síťový démon a zda se má po měření automaticky zakázat a zamaskovat. Získal jsem časovou úsporu cca 1 s.

Službu systemd-timesyncd jsem optimalizoval a při staru jsem tak ušetřil přibližně 0,5 s. Službu mi stačilo zakázat. Výsledkem ale nebylo pouze zrychlení startu. Přišel jsem o časovou synchronizaci. Hodiny systému tedy neodpovídaly korektnímu světovému času. To se i projevilo na odeslaných datech. Tento čas jsem se rozhodl neřešit.

V budoucnu bych buď doplnil skript o tuto funkci či přesunul v tomto případě vložení časového údaje na stranu serveru.

V případě, že cílový systém může přijít o funkci překladu záznamů skrze externí server DNS, můžeme zakázat službu systemd-resolved. Zrychlení startu se pak pohybovalo okolo 0,7 s. Vliv na komunikaci založenou na adresách IP tato optimalizace neměla. Při odesílání dat jsem adresu definoval jako konkrétní IPv4 adresu, proto jsem nemusel řešit část skriptu řešící odeslání dat.

Zamaskováním služby systemd-journald jsem přisel o nové logy a zároveň nedošlo k výraznému zrychlení. Proto jsem tuto optimalizaci vyhodnotil jako neefektivní a dále jsem ji nevěnoval pozornost. Po optimalizaci této služby jsem se také setkal se zamrznutím sytému. Došlo k tomu pouze jednou a pravděpodobně příčinou nebyla optimalizace. Nicméně by bylo nutné tuto chybu zduplikovat a analyzovat konkrétní příčinu z logů.

Službu tmp.mount jsem se rozhodl zamaskovat. Výsledkem měření byla úspora času, která někdy dosahovala až 0,3 s. Nicméně ztráta v podobě POSIX dočasného adresáře tmp mě dovedla k myšlence, že zrychlení není natolik markantní, abych musel odebírat důležitou komponentu systému souborů.

Polyfunkční systém mi přinesla optimalizace systemd-udevd. Abych například mohl používat řadu služeb, musel jsem si je nejprve nastartovat. Při startu nicméně tyto služby startují. Navíc optimalizace nepřinesla zásadní zrychlení startu. Zamaskování této služby jsem tedy nevyhodnotil pozitivně.

Optimalizací služby kmod-static-nodes jsem dokázal eliminovat náhodné výkyvy časových intervalů při startu. Bohužel eliminace výkyvů byla výsledkem jak zrychlení určitých služeb, tak ale i prodloužení startů služeb jiných. Optimalizaci jsem konkrétně realizoval zamaskováním služby.

Služba bluetooth, která se stará o funkci systému Bluetooth, je v řadě případů užití postradatelná. Proto jsem chtěl tuto službu rozhodně při staru zakázat. Konkrétně jsem službu musel zamaskovat, jelikož byla použita při startu i pokud byla zakázána. Ušetřil jsem tak 0,1 s při staru.

Optimalizace služby systemd-rfkill nevedla k markantnímu zrychlení startu. Opět jsem se po optimalizaci setkal s jedním náhodným zamrznutím systému. Kterému jsem opět nevěnoval pozornost, jelikož při následovném restartu opět systém nastartoval v pořádku. K optimalizaci jsem použil jak zakázání, tak i zamaskování služby.

Další optimalizace, která odstranila náhodné výkyvy časových intervalů startů OS, bylo zamaskování dev-mqueue.mount. Jak tomu ale bylo i v předešlém případě, opět se nejednalo pouze o zrychlení startu ostatních služeb, ale i o prodloužení startů řady dalších systémových služeb. Zejména pak u restartů vyvolaných příkazem reboot.

Poslední službou, které jsem se rozhod věnovat, byla systemd-hostnamed. Tato služba se i přesto, že jsem ji zamaskoval, po dvou restartech sama spustila. Příčinu jsem nehledal. V případech, kdy služba nenastartovala, došlo k ušetření času startu a to konkrétně 0,3 s.

3.2.3 Kombinování služeb

Dále jsem tedy pokračoval kombinováním jednotlivých optimalizací služeb. Snažil jsme se postupně identifkovat, zda mohou kombinace služeb vždy přinést zrychlení. Mé obavy z prodloužení startovací doby byly však mylné. Při použití služeb, které jsem vyhodnotil jako vhodné pro optimalizaci, jsem dosáhl výsledku, který výrazně zrychlil start OS. U této volby jsem vycházel vždy ze zdraví systému a z dostatečného optimalizačního přínosu.

Užitečnou kombinací například byly služby pro bezdrátové sítě a mobilní zařízení. Taktéž jsem rychlost startu OS zvýšil kombinací služeb, které souvisí se síťovým provozem. Spojení těchto dvou kombinací bylo jasným vítězem a tvoří základy většiny mých optimalizací. Taktéž jsem je zahrnul do finální výsledné kombinace. Ovšem pokud bych například systém zaváděl ze síťového disku, jistě bych musel kombinace upravit. V případě, kdybych mapoval nedosažitelné souborové systémy a diskové oddíly, čas startu by se markantně navýšil. Systém by prováděl řadu pokusů o připojení a zároveň by musel pro správnou identifikaci nedosažitelnosti čekat.

3.3 Volba funkcí a modulů jádra OS

U práce s jádrem a jeho konfigurací jsem využil obdobné měření jako v případě služeb. Měřil jsem tedy SW, HW restarty a první start OS. Mimo volby optimální konfigurace jádra jsem také volil modly, které budou použity. Tabulka 1 byla pro mě výchozím ukazatelem modulů, které bylo třeba odebrat. Zároveň jsem taktéž pracoval s parametry vrstvy RPI. Tuto optimalizaci jsem se rozhod zařadit do této kapitoly, jelikož se jedná o typ optimalizace, kterou je nutné provést skrze projekt Yocto před kompilací jádra. Optimalizace v oblasti jádra pro mě vždy znamenala čistou kombinaci a nasazení. Celkově tedy tyto optimalizace zabraly více času na realizaci.

3.3.1 Optimalizace vrstvy RPI

První optimalizace, kterou jsem zkusil praktikova,t bylo zakázání úvodní startovací obrazovky s grafickou podobou duhy (splash screen). Konkrétně jsem do souboru local.conf dopsal řádek s obsahem DISABLE_SPLASH="1". Při startu se tedy duhová obrazovka nezobrazila a já jsem tak získal průměrné zrychlení startu 0,3 s.

Zkusil jsem také nastavit minimální paměť GPU. V tomto případě jsem doplnil soubor local.conf o GPU_MEM="16". Výsledek byl pro mne překvapením, jelikož došlo naopak k průměrnému nárůstu doby startu. Další odlišné chování systému jsem nevypozoroval.

Dopsáním řádku DISABLE_OVERSCAN="1" do souboru local.conf jsem testoval odebrání podpory přizpůsobení okna obrazovky pro RPI. Po aplikaci se systém choval standardně a obraz se prakticky nezměnil. To bylo žádoucí, avšak nedošlo také k výraznému zrychlení startu. Otázkou by zde bylo, jak by se systém choval, pokud by se k jeho ovládání využívalo GUI. V tomto ohledu se naskytuje i možnost testování různých grafických prostředí pro OS Linux.

Položka BOOT_DELAY, kterou jsem nastavil na 0, v mém případě neznamenala zrychlení. Pravděpodobně má konfigurace obrazu OS v základu používá hodnotu 0, aniž by se tato skutečnost musel nastavit. Pokud jsem BOOT_DELAY naopak nastavil na hodnotu větší než 0, došlo k prodloužení startu.

Vrstva poskytuje možnost přetaktování, které by mělo primárně zvýšit rychlost prováděných akcí systému. Pasivně by tak tedy mělo dojít ke zrychlení startu. Proto jsem tuto praktiku otestoval a získal jsem potvrzující výsledek. Ušetřil jsem v průměru 0,3 s. Mimo zrychlení jsem také zaznamenal nárůst teploty CPU a RAM. Také mělo zařízení vyšší spotřebu, což by znamenalo problém pro zařízení napájené z baterie. Do local.conf jsem zapsal přetaktování frekvence procesoru řádkem ARM_FREQ="1550" a základní frekvenci jádra GPU díky dopsání CORE_FREQ="500". Navýšil jsem frekvenci čipu SDRAM , která udává rychlost pro RAM (SDRAM_FREQ="550"). Také jsem musel zvýšit napětí řádkem OVER_VOLTAGE="8". Tyto hodnoty by šlo dále navyšovat, a to do hranic jednotlivých komponent.

Poslední optimalizací vrstvy RPI bylo zakázaní zobrazení startovacího loga RPI. Jedná se o odebrání vykreslení řádku grafických objektů, které vypadají jako maliny. Zákaz jsem opět provedl úpravou local.conf, kdy jsem soubor rozšířil o řádek s údajem DISABLE_RPI_BOOT_LOGO="1". Časová úspora byla v mém případě v průměru 0,1 s. Vizuálně systém při startu nevykresloval nic jiného, pouze symbol _ a černou obrazovku, kdy jsem čekal na dokončení startovací fáze.

Do výsledné optimalizace vrstvy RPI jsem tedy zahrnul zakázání úvodní startovací obrazovky s grafickou podobou duhy, přetaktování a zvýšení výkonu a odebrání loga RPI při startu. Taktéž jsem vyzkoušel všechny zmíněné optimalizace a postupoval jsem komparativně. Porovnal jsem tedy výslednou optimalizaci a tu, kde jsem zahrnul všechny optimalizace. Data jednoznačně prokázala, že start systému využívající výslednou optimalizaci, byl rychlejší.

3.3.2 Optimalizace jednotlivých modulů

Tabulka 1 popisuje jednotlivé moduly, které jsem v systému měl zavedeny před optimalizací a se kterými jsem mohl pracovat. U volby modulů jsem nejčastěji pracoval s těmi, které jsou určeny pro bezdrátové technologie. V práci tedy popisuji postup, který je vhodný pro zařízení, která nevyužívají bezdrátovou komunikaci. Čistě bezdrátové zařízení by například nemělo fyzické komunikační porty. V takovém případě by bylo vhodné se zaměřit naopak na odebrání podpory právě takových fyzických komunikačních médií, jako jsou porty a sběrnice.

Interaktivně jsem v programu menuconfig vyhledal podporu síťování. V ní jsem odebral položku označenou jako podpora subsystému Bluetooth. Po kompilaci následované nasazením jsem zjistil, že došlo k odebrání modulů rfcomm, bnep, hci_uart, btbcm, serdev, bluetooth, ecdh_generic. Časová úspora této optimalizace se pak pohybovala okolo 1 s. V repozitáři je recept zajišťující konfiguraci modulů uložen jako soubor configs/recipes/no-bt. Velikost obrazu OS se zmenšila velice málo, jelikož mi analýza zobrazila zaokrouhlenou velikost jako 196 MB.

Odebrání ostatních bezdrátových sítí jako například Wi-Fi jsem nakonfiguroval v položce bezdrátové síťování. Odebral jsem moduly brcmfmac, brcmutil, cfg80211, rfkill. Výsledný fragment jsem v repozitáři umístil do souboru configs/recipes/no-wireless. Vypozoroval jsem zmenšení výsledného obrazu OS s výslednou velikostí 188 MB. Dopad na zrychlení systému byl pozitivní a ušetřil jsem přibližně 2 s času startu.

Mírné zrychlení přineslo taktéž odebrání podpory protokolu IPv6. V menuconfig jsem vyhledal podporu síťování, následně možnosti síťování a zde se nacházela volba pro protokol IPv6. Odebrání modulu prakticky skoro nezměnilo velikost výsledného obrazu OS. Tuto optimalizaci bude v budoucnu pravděpodobně nutno vyřadit. Důvodem by mohlo být převzetí dominance protokolu IPv6 nad IPv4. Vygenerovaný fragment jsem uložil do souboru configs/recipes/no-ipv6.

Na závěr jsem tedy hromadně odebral nadbytečné moduly, které jsem uvedl výše. Protože měly vliv na start všechny, odebral jsem všechny. Toto odebrání jsem realizoval spojením externě uložených fragmetnů no-bts, no-wireless a no-ipv6. Taktéž jsem zahrnul ještě dvě konfigurace jádra uložené ve fragmentech no-debug a lzo.

3.3.3 Ostatní optimalizace

První doplňkovou konfigurací jádra byla změna komprese jádra OS. V základu Linux využívá kompresi Gzip, která je výhodná zejména díky rychlé kompresi. Mezi její nevýhody se může řadit větší paměťový otisk. Celkově je tedy Gzip vhodné zejména pro zařízení, která jsou limitována malou pamětí. Já se rozhodl Gzip nahradit ztrátovou kompresí LZO. Díky ní jsem mohl zajistit rychlejší dekompresi jádra na úkor požadavku na větší paměť. Kompresní poměr byl větší až o 10 %. Fragment konfigurace jsem uložil

do souboru configs/recipes/lzo. Výsledek měření pak ukázal zrychlení HW startů o 1 s v průměru a zpomalení SW startu o 1 s. Průměrně zrychlení všech startů se pak pohybovalo okolo 0,3 s.

Posledním nastavením jádra, které jsem se rozhodl provést, bylo odebrání ladicí podpory jádra. Ušetřil jsem tak při startu 2 s. Tuto optimalizaci jsem musel provést až jako poslední, protože jinak bych nemohl využít ladicí nástroje a v případě problému by bylo obtížné jej identifikovat či vůbec zaregistrovat. Fragment je k dispozici v souboru configs/recipes/no-debug.

4 Diskuse nad výsledky

Jak ukázala analýza na naměřených průkazných datech, reálné zrychlení startu neumožnily všechny navržené optimalizace. Některé optimalizace měly sporné výsledky, kdy změna v průměrném časovém intervalu byla tak malá, že nebylo možné jednoznačně určit, zda rozdíl ve výsledných hodnotách nebyl ovlivněn jinými faktory. Veškeré výsledky práce je možné interaktivně projít na adrese http://app.dlouhybp.tkkzmj.cz.

Neoptimalizovaný systém nastartoval nejrychleji za 12,6 s. Veškeré optimalizace jsem porovnával s tímto startem, a i konečnou výslednou optimalizaci, která zahrnuje optimalizace s nejlepšími výsledky měření. Takovému systému trval nejrychlejší start pouze 4,2 s. Ve výsledku jsem tedy start OS zrychlil o 8,4 s výběrem vhodných služeb, výběrem modulů, konfigurací jádra a konfigurací vrstvy RPI. Takový systém bych doporučil použít v případě, kdy čtenář využívá jako HW platformu Raspberry Pi 3B+, je mu jedno, jakou distribuci OS použije a má znalosti projektu Yocto. Zároveň ví, že nepotřebuje pracovat s bezdrátovými technologiemi, které zařízení poskytuje. Jedná se o obecnou optimalizaci systému. Pokud by byl znám přesný požadavek a přesný scénář užití, pravděpodobně by se doba startu mohla snížit ještě více, a to různými praktikami, které jsem v práci zmínil, ale rozhodl jsem se je neaplikovat.

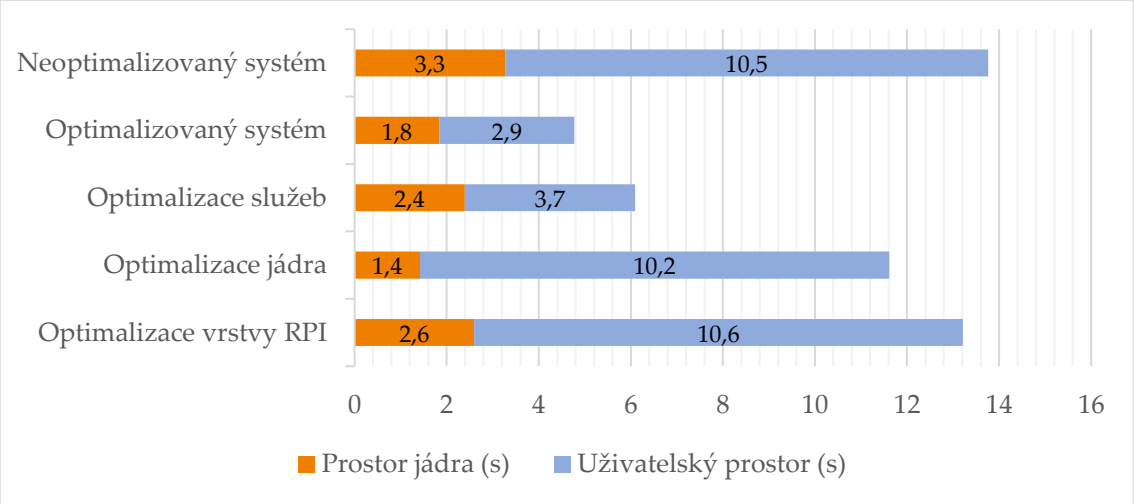
Mezi optimalizace, kterých lze snadno dosáhnout, bych zařadil optimalizace systémových služeb. Stačí k nim základní znalost práce s OS Linux a se službami tohoto OS. Navíc je možné tyto optimalizace snadno provádět na již existujícím systému. Měření u těchto optimalizací probíhalo rychleji. Vždy mi stačilo pouze nasadit čistý obraz OS, který jsem nemusel opět sestavovat a přeložit. Navíc tato optimalizace také přinesla nejvyšší míru zrychlení startu OS. Nejefektivnější optimalizací, kterou jsem zde provedl bylo zakázání služby hciuart, kdy jsem ušetřil v průměru až 6,5 s. Proto bych doporučil při optimalizaci začít vždy u systémových služeb. Ve výsledku optimalizovaný systém nejrychleji naběhl za 6 s.

Optimalizace, které již potřebují minimálně znalost vývoje jádra operačního systému, jsou volba modulů jádra a jeho konfigurace. Já jsem používal pro optimalizování vývoj za pomoci projektu Yocto. Čtenář si tak může snadno sám vyzkoušet provedené

optimalizace, či navázat na tuto práci. Výsledné hodnoty zrychlení u této optimalizace nebyly tak vysoké jako u služeb. Nejrychlejší start zde trval 11,3 s. Proto bych doporučil optimalizaci pouze v případě, pokud chce čtenář co nejmenší velikost jádra a ví, že funkcionality poskytnuté odebranými moduly nebude potřebovat ani výhledově v budoucnosti.

Obdobně je tomu tak i u konfigurace vrstvy RPI. Zde trval nejrychlejší start 12,8 s, což je na první pohled vyšší hodnota než nejrychlejší start neoptimalizovaného systému. Proto jsem provedl komparaci průměrů jednotlivých startů optimalizovaného a neoptimalizovaného systému. V průměru pak byl systém optimalizovaný opravdu rychlejší. Došlo totiž k vyrovnání jednotlivých intervalů startu optimalizovaného systému. Je třeba, aby si čtenář uvědomil, že tato optimalizace je primárně vázána na projekt Yocto. To znamená nutnost znalosti vrstev a receptů. Současně při použití jiného nástroje, než projektu Yocto při vývoji, bude třeba optimalizaci realizovat jiným způsobem.

Obrázek 13 zobrazuje porovnání jednotlivých shrnujících optimalizací. Uvedené hodnoty jsou v sekundách a pro každý start je vykreslena ve sloupci vždy hodnota, kterou OS strávil při startu v uživatelském prostoru a prostoru jádra. Jedná se o průměr naměřených hodnot startu dané optimalizace. Jak je vidět, optimalizace jádra a vrstvy RPI vedly k úspoře času zejména v oblasti prostoru jádra. Uživatelský prostor bezkonkurečně zrychlila optimalizace služeb. Tento výsledek taktéž ukazuje, jakým způsobem můžeme start zrychlit, když víme, v jakém prostoru tráví systém při startu nejvíce času.



Obrázek 13: Komparace optimalizací

5 Závěr

Cíle mé práce vyžadovaly popis startu, návrh a nasazení optimalizace a volbu nejlepší optimalizace. Všechny tyto cíle jsem úspěšně splnil. V práci jsem detailně popsal proces startu. Uvedl jsem několik možných optimalizací. Následně jsem vytvořil ekosystém, kterým jsem vyřešil problematiku sběru dat pomocí konfigurovatelného skriptu. Ekosystémem řeším také požadavek na zpracování nasnímaných dat backend aplikací, která data ukládá v NoSQL databázi MongoDB. Aplikace také poskytuje uložená data pro frontend aplikaci, kterou jsem vytvořil pro vizualizaci data a pro datovou analýzu. Mohl jsem tak snadno zahrnout do této práce výsledné zhodnocení jednotlivých optimalizací a vyvození výsledné optimalizace. Vzhledem k tomu, že jsou tyto součásti volně přístupné, poskytl jsem i naměřená data pro další využití. Čtenáři jsem tak dal možnost si interaktivně prohlédnout data z celkem 130 sestavení obrazů operačních systémů a z celkem 566 startů těchto sestavení.

Celá práce je tvořena na možnost budoucího využití, jelikož bude začleněna do projektu BusKit. Projekt je realizován TUL a řeší protokol pro síťovou komunikaci uvnitř automobilu. Proto jsem do práce zahrnul taktéž dokumentaci a jednotlivé komponenty ekosystému jsem připravil na flexibilní nasazení a využití včetně vývoje.

Možné budoucí rozšíření této práce vidím v optimalizování dalších míst, která nebyla součástí zadání. Například optimalizací HW, zavaděče či testování volby jiného média. Také si myslím, že je možné navázat na automatizaci procesu vývoje a nasazení. Celý můj ekosystém je taktéž možné dále vyvíjet.

Použitá literatura

[1] SIMMONDS, Chris. *Mastering Embedded Linux Programming*. First published. Birmingham: Packt Publishing Ltd., 2015. ISBN 978-1-78439-253-6.

[2] LOVE, Robert. *Linux kernel development*. Third edition. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN 978-0-672-32946-3.

[3] GNU GRUB. In: *GNU* [online]. Boston: Free Software Foundation, 2022 [cit. 2022-05-08]. Dostupné z: https://www.gnu.org/software/grub/index.html

[4] MOSS, Bob. Make Linux faster, lighter and more powerful: Optimise the boot process, KDE, Gnome, networking and more. In: *TechRadar: The source for tech buying advice* [online]. New York: Future US, Inc., 2022 [cit. 2022-05-08]. Dostupné z: https://www.techradar.com/in/news/computing/pc/make-linux-faster-lighter-and-more-powerful-641317

[5] MURRAY, Andrew. 12 Lessons Learnt in Boot Time Reduction. In: *SlideShare* [online]. San Francisco: Scribd Inc., 2022 [cit. 2022-05-08]. Dostupné z: https://www.slideshare.net/andrewmurraympc/12-lessons-learnt-in-boot-time-reduction

[6] Embedded Linux boot time optimization training. In: *Bootlin: Embedded Linux and kernel engineering* [online]. Oullins: Bootlin, c2004-2022 [cit. 2022-05-08]. Dostupné z: https://bootlin.com/doc/training/boot-time/boot-time-slides.pdf

[7] JANEČEK, Vladislav. Vyzkoušeli jsme mikropočítač Raspberry Pi. In: *Živě.cz: O počítačích, IT a internetu* [online]. Praha: CZECH NEWS CENTER, 2022 [cit. 2022-05-08]. Dostupné z: https://www.zive.cz/clanky/vyzkouseli-jsme-mikropocitac-raspberry-pi/sc-3-a-165391/default.aspx

[8] Raspberry Pi Documentation. In: *Https://www.raspberrypi.com/* [online]. Londýn: Raspberry Pi Ltd, c2012-2022 [cit. 2022-05-08]. Dostupné z: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

[9] Installation. In: *GitLab Docs* [online]. Eindhoven: GitLab B.V., 2022 [cit. 2022-05-08]. Dostupné z: https://docs.gitlab.com/ee/install

[10] *Sphinx documentation* [online]. Georg Brandl and the Sphinx team, c2007-2022 [cit. 2022-05-08]. Dostupné z: https://www.sphinx-doc.org/en/master/

[11] PURDIE, Richard, Chris LARSON a Phil BLUNDELL. BitBake User Manual. In: *Yocto Project* [online]. California: Yocto Project, 2022 [cit. 2022-05-08]. Dostupné z: https://www.yoctoproject.org/docs/1.6.1/bitbake-user-manual/bitbake-user-manual.html

[12] *OpenEmbedded.org* [online]. OpenEmbedded, 2017 [cit. 2022-05-08]. Dostupné z: https://www.openembedded.org/wiki/Main_Page

[13] 4 Yocto Project Concepts. In: *Welcome to the Yocto Project Documentation* [online]. The Linux Foundation, c2010-2022 [cit. 2022-05-08]. Dostupné z: https://docs.yoctoproject.org/overview-manual/concepts.html

[14] Embedded System Market. In: *Https://www.gminsights.com/* [online]. Delaware: Global Market Insights Inc., 2022 [cit. 2022-05-08]. Dostupné z: https://www.gminsights.com/industry-analysis/embedded-system-market

[15] Getting Started: React. In: *React: A JavaScript library for building user interfaces* [online]. Meta Platforms, 2022 [cit. 2022-05-08]. Dostupné z: https://reactjs.org/docs/getting-started.html

[16] *Yocto Project* [online]. California: Yocto Project, 2022 [cit. 2022-05-08]. Dostupné z: https://www.yoctoproject.org/

[17] Top 4 Embedded Operating Systems with Examples (2022 Update). In: *Felgo* [online]. Vídeň: Felgo, 2022 [cit. 2022-05-08]. Dostupné z: https://blog.felgo.com/embedded/embedded-operating-systems

Seznam příloh

Tato práce neobsahuje žádné přílohy.