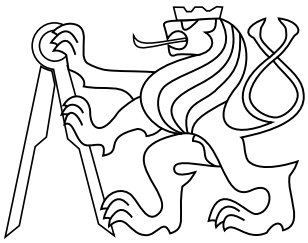
bakalářská práce

Automatická analýza vlivu konfigurace Linuxu na výkonnost systému

Karel Kočí



Květen 2015

Ing. Michal Sojka, Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická, Katedra řízení

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Karel Kočí**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Automatická analýza vlivu konfigurace Linuxu na výkonnost systému**

Pokyny pro vypracování:

1. Seznamte se se systémem Kconfig pro konfiguraci Linuxového jádra a nástroji pro řešení SAT problémů.2. Vytvořte software, který umožní najít všechny přípustné kombinace konfiguračních voleb Linuxu vzhledem k ručně daným omezením.3. Pro všechny nalezené konfigurace Linux zkompilujte, nabootujte a spusťte sadu benchmarků.4. Navrhněte metodu jak z měřených dat odhadovat vliv jednotlivých konfiguračních voleb na celkovou výkonnost systému.5. Vše důkladně zdokumentujte.

Seznam odborné literatury:

[1] Tartler, Reinhard ; Lohmann, Daniel ; Sincero, Julio ; Schröder-Preikschat, Wolfgang: Feature Consistency in Compile-Time Configurable System Software . In: Heiser, Gernoth ; Kirsch, Christoph (Ed.) : Proceedings of the EuroSys 2011 Conference (EuroSys '11) (EuroSys 2011 Salzburg 10-13 April 2011). New York, NY, USA : ACM, 2011, pp 47-60. - ISBN 978-1-4503-0634-8[2] Tartler, Reinhard ; Lohmann, Daniel ; Dietrich, Christian ; Egger, Christoph ; Sincero, Julio: Configuration coverage in the analysis of large-scale system software . In: SIGOPS Oper. Syst. Rev. (ACM OSR) 45 (2012), No. 3, pp 10-14

Vedoucí: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 20. 2. 2015

Poděkování

Na tomto místě bych chtěl poděkovat Ing. Michalu Sojkovi, Ph.D za odborné vedení mé bakalářské práce, za trpělivost a konzultace k jejímu obsahu. Také bych rád poděkoval své rodině a přátelům, že mi poskytli dostatek prostoru a podpory k její vypracování.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Abstrakt

Jádro operačního systému Linux obsahuje mnoho různých subsystémů a ovladačů pro zařízení. Před překladem ze zdrojových kódů se vybírá, které subsystémy a ovladače mají být zahrnuty, tzv. konfiguruje se. Jeho konfigurace není triviální. Existují různé nástroje, které se snaží různě tuto konfigurovatelnost analyzovat a vyhledávat chyby a problémy. Cílem této práce je ověřit, zdali je možné zjistit vliv vybraných konfiguračních voleb na celkovou výkonnost systému měřením výkonu různých konfigurací. Za tímto účelem byly vytvořeny nástroje pro automatické měření a analýzu výkonu generovaných konfigurací. Výsledky získané tímto nástrojem ovšem, jak se zjistilo, nejsou průkazné.

Klíčová slova

Linux; Kconfig; SAT solver; výkonnost; automatická analýza

Abstrakt

Kernel of Linux operation system contains many different subsystems and drivers for various hardware. Configuration is done by selecting subsystems and drivers before building from source codes. But configuration is not trivial task. Various tools are trying to analyze such configurations and search for mistakes and problems. Goal of this work is to verify, that we can get impact on performance per configuration symbols by measuring whole system with different configurations. And for this purpose were created tools for automatic measuring and analysis of generated configurations performance. But results gained by that tool are not conclusive, as it was find out.

Keywords

Linux; Kconfig; SAT solver; performance; automatic analysis

Obsah

1 Úvod

Tato práce se zabývá Linuxem a konfigurací jeho překladu. Linux je jádro operačního systému. Je to projekt s otevřeným zdrojovým kódem s přispěvateli z celého světa. Je široce užíván v rozličných zařízeních. Pohání výkonná zařízení jako jsou servery, ale i domácí počítače, mobilní telefony až malá vestavěná zařízení. Je také dostupný pro širokou paletu procesorových architektur. Tuto rozličnost užití umožňuje vysoká konfigurovatelnost. Počet konfiguračních voleb pro jednu procesorovou architekturu se pohybuje v řádu tisíců (například něco okolo devíti tisíc pro architekturu x86). Takto velký počet konfiguračních voleb představuje nové výzvy pro udržitelnost kódu a testování. Vznikají například projekty pro optimalizaci konfigurace pro danou aplikaci [1], nebo projekty, které se pokoušejí automaticky hledat problémy v tak rozsáhlé konfiguraci [16].

Tato práce je zaměřena především na výkonnost systému. Různé konfigurace vedou na různé varianty výsledného jádra. Ty pak mají různou výkonnost. Výkonnost se dá měřit pomocí testů (benchmarků). Pro různé testy se může změna mezi variantami lišit.

V této práci se budu zabývat domněnkou, že je možné zjistit příspěvek jednotlivých konfiguračních voleb na celkovém výkonu jádra, a to pomocí měření různých konfigurací a následnou jednoduchou analýzou získaných dat.

Výsledné hodnoty mohou být dále prakticky použity. Znalost, jakým způsobem, která konfigurační volba ovlivňuje výkonnost, může posloužit například pro optimalizaci konfigurací. Nebo pro bližší diagnostikování problémů s výkonem.

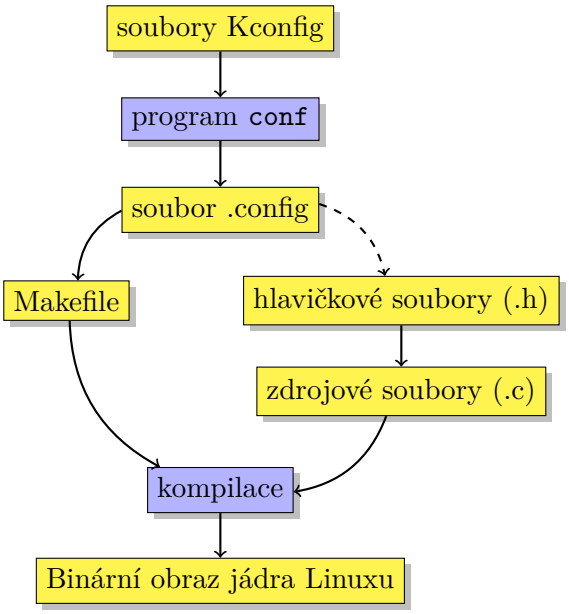
K tomuto účelu byly v rámci této práce vyvinuty nástroje, které mají umožnit automatický proces měření výkonnosti různých konfigurací. Generují se přípustné konfigurace a měří se jejich výkonnost. Naměřené hodnoty jsou pak automaticky vyhodnoceny.

2 Použité techniky a technologie

V této kapitole zmiňuji technologie a techniky, které jsou v práci využity. Píši o Linuxu a hlavně jeho konfigurování. Zmiňuji se o SAT problému o nástrojích Novaboot, Buildroot a SciPy.

2.1 Linux

Linux je jádro operačního systému. Je to projekt s otevřeným zdrojovým kódem. Jádro obsahuje mnoho různých subsystémů a velké množství ovladačů pro různý hardware. Zároveň Linux podporuje různé procesorové architektury. Linux je tak možné sestavit pro různé konfigurace a tedy pro různé účely. Velké množství možností ale vede k nepřehlednosti a složitosti konfigurace a následného překladu. Linux tento problém řeší sadou nástrojů označovanou souhrnně jako Kbuild. Ty mají za úkol umožnit uživateli vytvořit konfiguraci a na základě této konfigurace přeložit Linux. Část těchto nástrojů, která slouží ke konfiguraci, je označována jako Kconfig. Na obrázku 1 je blokově znázorněn zjednodušený postup konfigurace. Žlutou barvou jsou znázorněny soubory. Modrou barvou jsou znázorněny programy a programové operace.



Obrázek 1 Schéma konfigurace Linuxu

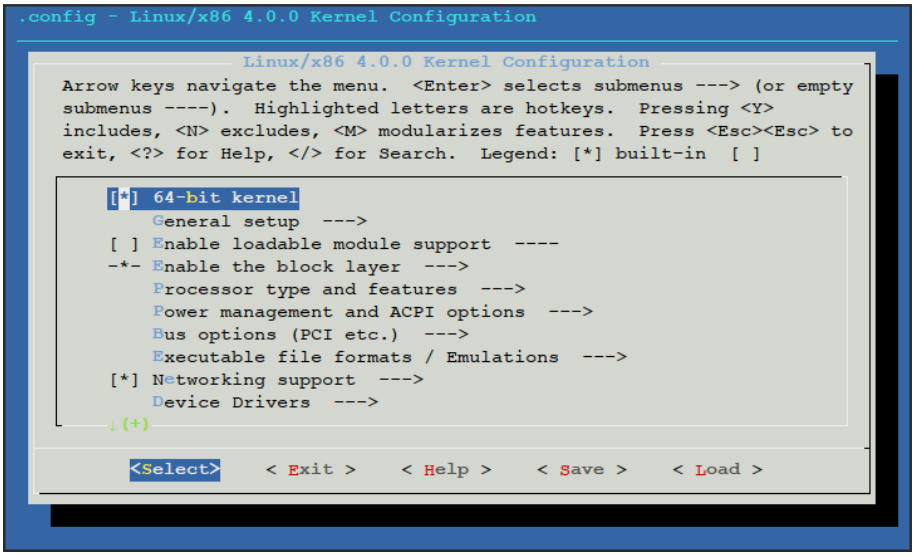
2.1.1 Kconfig

Kconfig je sada nástrojů a definice jazyka [2] vyvinutá za účelem konfigurace Linuxu před překladem ze zdrojových kódů. Konfigurací je myšleno ovlivňování překladu zdrojových kódů, a to definováním, které soubory budou přeloženy, a podmíněným překladem reprezentovaným „ifdef“ bloky v kódu.

Konfigurace probíhá pomocí přiřazení hodnot jednotlivým konfiguračním volbám. S výjimkami se dá prohlásit, že každá z voleb zastupuje nějaký kód nebo vlastnost programu. Můžeme se tak rozhodovat, zdali danou vlastnost požadujeme, nebo ne. V některých případech se navíc můžeme rozhodnout, zdali danou vlastnost, například ovladač k hardwaru, chceme mít přímo součástí jádra, nebo jako dynamický modul. Proto je Kconfig postaven na třístavové logice.

Volby jsou definovány v textových souborech pomocí jazyku Kconfig. Tyto soubory se většinou jmenují „Kconfig“, ale není to podmínkou. Celková konfigurace vzniká spojením jednotlivých Kconfig souborů. Díky tomu může být definice konfigurace ve stejném adresáři jako související zdrojové kódy.

Jednotlivé konfigurační volby se skládají do tzv. menu, přičemž struktura menu usnadňuje orientaci v konfiguračních volbách. Každé menu může mimo jiné také obsahovat další podmenu. Tuto stromovou strukturu využívají především grafické programy [3]. Příkladem programu, který menu využívá, je `mconf` na obrázku 2.



Obrázek 2 Interaktivní program `mconf` s hlavním menu

2.1.2 Jazyk Kconfig

Jedná se o jazyk složený z klíčových slov, jmen konfiguračních voleb a speciálních znaků. Jazyk umožňuje specifikovat jednotlivé konfigurační volby, jejich typ, jejich vzájemné závislosti a grafickou prezentaci v uživatelském rozhraní. Konfigurační volby jsou definovány názvem a atributy. Kromě graficky viditelných a uživatelsky nastavitelných voleb také mohou být definovány volby neviditelné, určené k jejich organizaci a zjednodušení závislostí. Tyto volby jsou automaticky nastavovány dle pravidel popsaných v podsekci 2.1.2 o atributu *default*. Obě kategorie voleb se zapisují stejně až na to, že volby, které nemají být viditelné a tak uživatelsky nastavitelné, nespecifikují atribut *prompt* (viz. 2.1.2).

V následujících odstavcích je popsána část specifikace jazyka [2]. Jsou zde vynechány části, které přímo nesouvisí s prací, klíčová slova, která reprezentují zkrácený zápis více atributů současně.

Specifikace každé konfigurační volby může mít několik řádků. První řádek vždy začíná klíčovým slovem *config*. Následuje název volby, většinou slovo či zkratka psaná velkým písmem a spojovaná podtržítky (tato konvence vychází z požadavku na zápis

symbolu pro C preprocessor). Na řádcích následujících po řádku s *config* jsou specifikovány atributy, přičemž text je od začátku řádku odsazen znakem tabulátor. Každý atribut začíná klíčovým slovem, které ho definuje. Jsou to klíčová slova udávající název typu a klíčová slova *default*, *prompt*, *depends on*, *select* a *visible if*. Příklad definice konfigurační volby je na obrázku 3.

```
config X86_EXTENDED_PLATFORM
        bool "Support for extended (non-PC) x86 platforms"
        default y
        depends on X86_64
```

Obrázek 3 Ukázka specifikace konfigurační volby

Třístavová logika, která je zde využívána, nabývá následujících hodnot: *y*, *n* a *m*. Nástroj Kconfig jim přiřazuje čísla pro matematický a interní zápis operací a to takto: $y = 2$, $m = 1$ a $n = 0$.

Před samotným popisem atributů nejprve definujeme pojmy, které budou použity.

Konfigurační volba je *zvolena*, pokud je její hodnota jiná než *n*.

Výraz je definován syntaktickým pravidlem na obrázku 4. První řádka říká, že každá konfigurační volba je výrazem. Druhá řádka definuje výraz porovnání hodnot konfiguračních voleb. Když se rovnají, je výsledná hodnota výrazu 2. Pokud se nerovnají, je výsledná hodnota 0. Třetí řádek definuje nerovnost dvou konfiguračních voleb. Když se nerovnají, je výsledná hodnota 2. V opačném případě je výsledná hodnota 0. Čtvrtý řádek říká, že mohou být využity závorky pro seskupení výrazů. Pátý řádek představuje negaci výrazu. Jeho hodnota je definována jako $2 - výraz$. Šestý řádek definuje logický součin a hodnota výsledného výrazu je definována jako minimum z hodnot jednotlivých podvýrazů výrazů. Poslední, sedmý řádek definuje logický součet a hodnota výsledného výrazu je maximum z hodnot jednotlivých podvýrazů výrazů.

```
<výraz> ::= <volba>           (1)
          <volba> = <volba>   (2)
          <volba> != <volba>  (3)
          ( <výraz> )         (4)
          ! <výraz>           (5)
          <výraz> && <výraz>  (6)
          <výraz> || <výraz>  (7)
```

Obrázek 4 Syntaxe výrazu [2]

Výraz je považován za *splněný*, pokud je jeho hodnota větší než nula.

Za některými atributy může být volitelně uváděn *podmínkový blok*. Připojuje se na konec řádku a začíná klíčovým slovem *if* a za ním následuje výraz. Pokud je výraz splněný, je atribut použit. V opačném případě je atribut ignorován. Ukázka zápisu je na obrázku 5.

Atributy typu

Prvním popsaným atributem je atribut definující typ konfigurační volby. Ten musí být specifikován u každé konfigurační volby, a to právě jednou. Je definován pomocí jedním z těchto klíčových slov:

```
config HPET_TIMER
        bool
        default X86_64
        prompt "HPET Timer Support" if X86_32
```

Obrázek 5 Ukázka použití podmínkového bloku u atributu *prompt*

- *bool*
- *tristate*
- *string*
- *int*
- *hex*

Typ *bool* nabývá právě dvou hodnot, ano (*y*) a ne (*n*). Typ *tristate* je podobný typu *bool*. Přidává hodnotu modul (*m*). Typ *string* představuje naopak řetězec znaků, například slova či věty. Zbylé dva typy jsou interně reprezentovány jako typ *string*. Jedná se o dva různé zápisy čísel. Celočíselný a obecně hexadecimální.

Za klíčovým slovem definující typ je možné dále ještě uvést do uvozovek text pro zobrazení, což zastupuje atribut *prompt*.

Atribut *default*

Výchozí hodnota konfigurační volby je definována pomocí atributu *default*, za nímž následuje specifikace výchozí hodnoty v souladu s definovaným typem. Konfigurační volba může mít výchozích hodnot více. O tom, která volba bude využita, pak rozhoduje podmínkový blok, který je možné za tento atribut připsat. Pokud bude definováno více výchozích hodnot, a to bez podmínkového bloku nebo se splněnými podmínkami, je z nich vždy vybrána dříve definovaná. Pokud není žádná výchozí hodnota specifikována, má typ *bool* a *tristate* výchozí hodnotu *n* a ostatní typy hodnotu prázdného řetězce. Pro číselné typy se tak jedná o nevalidní hodnotu.

Výchozí hodnota je použita, pokud uživatel danou volbu nezmění, nebo pokud k ní nemá přístup, protože není viditelná (např. proto, že nemá žádný text pro zobrazení). V takovém případě ale může být přepsána pomocí atributů *select* ostatních voleb.

Atribut *prompt*

Atribut *prompt*, za kterým následuje text v uvozovkách, určuje popisek, se kterým je konfigurační volba zobrazena v uživatelském rozhraní. Samotný text může být také specifikován za atributem typu, a tak nahradit tento atribut. Jedna volba může mít nejvíce jeden tento atribut. Atributu dále mohou být přidány závislosti pomocí podmínkového bloku, který je možný za tento atribut připsat.

Atribut *depends on*

Atribut *depends on* je výraz, který musí být splněn, aby bylo možné konfigurační volbu zvolit a byla viditelná v uživatelském rozhraní. Zápis závislostí se uvozuje klíčovým souslovím *depens on*, za nímž následuje výraz. Pokud je těchto atributů definováno pro jednu volbu více, jsou spojeny jednotlivé specifikované výrazy logickou konjunkcí.

Atribut *select*

Zápis tohoto atributu se uvozuje klíčovým slova *select*, za kterým následuje jméno cílové volby. Volitelně je možné za tento zápis připojit podmínkový blok. Atribut *select* zajišťuje, že cílová konfigurační volba má hodnotu minimálně stejnou, nebo vyšší než konfigurační volba, u které je tento atribut napsán. Vzhledem k tomu je možné tento atribut použít pouze na volby typu *bool* a *tristate*. Také se díky tomu označuje jako zpětná závislost.

Atribut *visible if*

Atribut *visible if*, za nímž následuje výraz, je možné specifikovat pouze u voleb s definovaným atributem *prompt*. Pokud je výraz splněný a je také splněn výraz atributu *depends on*, je umožněno uživateli změnit hodnotu volby.

Výběr z voleb

Kromě voleb uvozených klíčovým slovem *config*, je možné definovat výběr z několika voleb. Zajišťuje tak jednoduše volbu pouze jediné volby. Volbu definujeme tak, že na nový řádek zapíšeme klíčové slovo *choice*. Na řádcích po tomto klíčovém slovu, stejně jako v případě klasické volby, můžeme napsat tabulátorem odsazené atributy. Volba může být, ale na rozdíl od konfigurační volby, definované pomocí klíčového slova *config*, pouze typy *bool* a *tristate*. Pokud je typu *bool*, je možné zvolit pouze jedinou volbu. Pokud je její typ *tristate*, může mít pouze jediná volba hodnotu *y* a libovolný počet může mít hodnotu *m*, zbývající mají hodnotu *n*. Volby také umožňují přídavný atribut *optional*, který specifikuje že ani jedna volba nemusí být zvolena.

Volby, ze kterých se skládá výběr, se píší na řádky za definici a atributy výběru. Skupina se poté uzavře řádkou s klíčovým slovem *endchoice*. Zkrácený příklad je na obrázku 6.

```
choice
        prompt "Kernel compression mode"
        default KERNEL_GZIP
        depends on HAVE_KERNEL_GZIP || HAVE_KERNEL_BZIP2 || ...

config KERNEL_GZIP
        bool "Gzip"
        depends on HAVE_KERNEL_GZIP

config KERNEL_BZIP2
        bool "Bzip2"
        depends on HAVE_KERNEL_BZIP2

...

endchoice
```

Obrázek 6 Zkrácený příklad definice výběru z voleb [4]

Menu

Konfigurační volby i výběry z voleb mohou být následně spojovány do společných sekcí, tzv. menu. Usnadňují tak orientaci uživateli v konfiguračních volbách. Menu je definováno pomocí klíčového slova *menu*, za které je do uvozovek vepsán popisek, pod kterým je konfigurační volba zobrazena v uživatelském rozhraní. Na následujícím řádku se může definovat atribut *depends on* a to stejně jako v případě konfigurační volby.

Volby a výběry z voleb obsažených v menu se vkládají na samostatné řádky za definici. Skupina se poté uzavře pomocí klíčového slova *endmenu*.

Všechny zahrnuté volby, výběry z voleb i menu získávají atribut *depends on* tak, jak je uveden v definici menu. Příklad je na obrázku 7. Konfigurační volba *NETDEVICES* tak má atribut *depends on NET*, i když u ní přímo není uveden.

```
menu "Network device support"
        depends on NET

config NETDEVICES
        ...

endmenu
```

Obrázek 7 Příklad definice menu [2]

2.1.3 Nástroje pro práci s jazykem Kconfig

Pro samotnou konfiguraci Linuxu existuje několik různých programů [3]. Nejvíce používané, jsou programy `conf` a `mconf`. Oba nabízejí rozhraní pro úpravu a vytváření konfigurací v závislosti na specifikaci v jazyku Kconfig a sdílejí tak společnou část kódu. Oba programy také komunikují s uživatelem pomocí vstupu a výstupu textového terminálu. Program `conf` vypisuje dotazy, uživatel na ně vkládá odpovědi, a tak specifikuje či upravuje konfiguraci. Program `mconf` naopak umožňuje plně interaktivní procházení konfiguračním menu. Díky těmto rozdílům je `conf` využíván převážně skripty, nebo jako generátor specifických konfigurací. Často se také využívá k vytváření výchozích konfigurací.

Program `mconf` je postaven nad zjednodušenou a upravenou verzí programu Dialog [5]. Tento program vytváří za pomoci znaků a barev uživatelské rozhraní v terminálu. Využívá přitom speciálních módů terminálu, nemusí tak být podporovány všechny terminálové emulátory a je omezena i minimální velikost terminálu. V Kconfigu je upravená verze programu Dialog do formy knihovny pojmenované jako lxdialog.

Součástí Linuxu jsou ještě další programy určené pro práci s konfigurací. Jedná se o programy `qconf`, `gconf` a `nconf`. Program `nconf` je terminálový program postavený přímo nad knihovnou ncurse [6] a je tak podobný programu `mconf`. Zbylé dva jsou grafické programy a jejich rozdíl je především v použité grafické knihovně. Qt4 v případě `qconf`, GTK+2.0 v případě `gconf`. Vzhledem k tomu, že všechny tři programy pouze nabízejí jinou grafickou formu, v některých případech i méně funkcí než `mconf`, nebudou zde dále rozebírány.

Všechny výše zmíněné programy ukládají konfiguraci do souboru *.config*, také se ho při spuštění pokoušejí načíst. Jeho syntaxe je popsána dále. Tento soubor tak představuje samotnou konfiguraci. Dále všechny programy využívají několik proměnných prostředí.

Jednou z důležitějších s ohledem na křížový překlad je proměnná *ARCH*, která umožňuje specifikovat architekturu, pro kterou má být Linux přeložen.

Uživatel většinou nespouští výše zmíněné programy přímo, ale pomocí programu `make`, jež jako vstup využívá soubor *Makefile*, který následně samotné programy volá se správnými argumenty. Program `mconf` se tak spouští takto: `make menuconfig`. Program `conf` je spouštěn větším počtem takovýchto příkazů, vždy za jiným cílem. Tak například `make defconfig` vyvolává `conf` za účelem vygenerování výchozí konfigurace. Dalšími příklady jsou `make allyesconfig` a `make allnoconfig`. Důležitým je také `make oldconfig`, který se využívá těsně přes spuštěním překladu.

2.1.4 Formát souboru konfigurace

Samotná konfigurace je ukládána ve vlastním souborovém formátu s jednoduchou syntaxí. Součástí syntaxe je, že každá zapisovaná volba se zapisuje s prefixem *CONFIG_*. Například konfigurační volba *X86* se tak zapíše jako *CONFIG_X86*. Jednotlivé konfigurační volby se píší na samostatné řádky a od nastavené hodnoty jsou odděleny znakem =. Pro typy *bool* a *tristate* se hodnota zapisuje písmeny *y*, *m* a *n*. Písmena odpovídají hodnotám, jak jsou zavedeny v sekci 2.1.2 o jazyku Kconfig. Vzhledem k tomu, že ostatní typy jsou přímo řetězce, jsou za rovnítkem uvedeny přímo a uzavřeny do uvozovek. Zvláštností v syntaxi je, žepřesto že za znakem „#“ následuje komentář a měl by tak být ignorován, tak tomu v některých případech není. V případě, že komentář začíná jménem konfigurační volby s prefixem a za ní následuje text *is not set*, je tato konfigurační volba načtena s informací, že nebyla uživatelsky změněna. Přestože tedy může být očekáváno, že taková volba není nastavena a tedy má hodnotu *n*, nemusí tomu tak být vždy. A to pokud má jinou výchozí hodnotu než *n* nebo pokud se vyskytnou zpětné závislosti od jiné konfigurační volby.

Do souboru se zapisují pouze ty konfigurační volby, které mají splněné výrazy atributu *depends on*.

2.2 SAT problém

SAT problém je problém zjištění, zdali existuje taková kombinace hodnot proměnných, pro kterou je daná booleovská formule pravdivá. Pokud taková kombinace existuje, je booleovská formule tzv. splnitelná [7]. Ověření, zdali je pro danou kombinaci proměnných booleovská formule splnitelná je snadné. Nalezení takové kombinace a tím dokázání, že existuje, je ale obtížné. Je dokázáno, že tento úkol je NP-úplný [8]. Existují různé implementace, které se pokoušejí tento problém řešit. Jednou z takových je zde použitý a uvedený program PicoSAT.

2.2.1 PicoSAT

PicoSAT [9] je program s celkem malou implementací. Hlavní implementace je napsána v jediném souboru. I přesto se jedná o dostatečně silný nástroj pro různé úkoly. V této práci byl vybrán především, protože umožňuje přímo generovat všechna možná řešení a to pomocí parametru `--all`.

Formát vstupních dat

PicoSAT vyžaduje, aby vstupní formule byla v konjunktivní normální formě. Tedy aby jednotlivé proměnné byly spojeny logickým součtem (disjunkcí) do klauzulí a ty se

spojují konjunkcí do výsledné formule. K praktickému zápisu se využívá zjednodušený souborový formát DIMACS CNF [10]. Ten je definován tak, že každá řádka začínající písmenem *c* je ignorována. První řádka, která není ignorována, musí začínat textem *p cnf*, za kterým následuje mezerou oddělený počet proměnných, a za další mezerou počet klauzulí. Definuje tak hlavičku souboru. Každý následující řádek definuje jednu klauzuli. Jsou to čísla oddělené mezerami. Přičemž proměnné jsou zapsány jako čísla. A jednička tak například představuje první proměnnou. Negace se definuje záporným číslem. Každá klauzule musí být nakonec ukončena nulou. Příklad souboru ve formátu DIMACS CNF je uveden na obrázku 8.

```
c (x1 OR NOT(x5) OR x4) AND
c (NOT(x1) OR x5 OR x3 OR x4) AND
c (NOT(x3) OR NOT(x4))
p cnf 5 3
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

Obrázek 8 Ukázka formátu vstupních dat pro PicoSAT

Formát výstupních dat

PicoSAT generuje výstup v jednom ze dvou podporovaných formátů. Formát, který bude použit, se určuje na základě argumentu na příkazové řádce. První formát je stejný jako vstupní formát. Druhý je definován tak, že každý řádek vždy začíná jedním ze tří možných písmen:

- *c* pro informativní a statistický výstup.
- *s*, za kterým následuje slovo *SATISFIABLE*, pokud je daná formule splnitelná, v opačném případě následuje slovo *UNSATISFIABLE*.
- Pokud řádek začíná písmenem *v*, jedná se o řádky s řešením. Hodnoty na tomto řádku jsou proměnné, pro záporné číslo je proměnná v řešení nepravdivá a pro kladné číslo je naopak proměnná pravdivá. Stejně jako v případě formátu vstupních dat je i řešení ukončeno nulou.

Podle počtu argumentů „-v“ na příkazové řádce PicoSAT generuje různé úrovně statistických a informačních výpisů. Pokud není žádný tento argument uveden, jsou všechny statistické a informační výpisy vypsány. Taková ukázka je na obrázku 9.

```
s SATISFIABLE
v 1 -2 -3 -4 5 0
```

Obrázek 9 Ukázka výstupu pro vstupní data uvedené na obrázku 8

2.3 Novaboot

Novaboot je nástroj, který zjednodušuje bootování operačního systému z jednoho počítače na jiný. Automaticky zajišťuje přesun potřebných souborů, generování konfigurací pro bootloader, resetování cílového počítače či přesměrování sériové linky na standardní

vstup/výstup. Chování Novabootu se nastavuje pomocí parametrů při spuštění, obsahu novaboot scriptu a konfiguračních souborů [11].

Vstupem novabootu je script, který může být také považován za zobecněnou konfiguraci bootloaderu. Script má definovanou jednoduchou syntaxi [12]. Čte se po řádcích, přičemž řádky začínající znakem „#“ jsou ignorovány. Ostatní řádky začínají buďto slovem *load*, *run* nebo *uboot*, pokud nejsou prázdné. Řádky se slovem *load* představují modul k nahrání do paměti bootovaného počítače a zbytek řádky potom jsou vloženy jako argumenty příkazové řádky. Slovo *run* uvozuje za ním následující příkaz spouštěný pomocí shellu (*/bin/sh*). Řádky začínající slovem *uboot* jsou příkazy pro zavaděč U-Boot. Příklad jednoduchého Novaboot skriptu je na obrázku 10. Tento script zajišťuje spuštění *make -C buildroot*, nahrání jádra *bzImage* a nahrání obrazu disku. K nabootování je zapotřebí již jen spustit tento skript a jako parametr mu předat cíl, kde chceme systém nabootovat. Například pro virtualizovaný systém je možné předat parametr *--qemu* [11].

```
#!/usr/bin/env novaboot
run  make -C buildroot
load bzImage console=ttyS0,115200
load buildroot/images/rootfs.cpio
```

Obrázek 10 Ukázka Novaboot skriptu [12]

2.4 Buildroot

Buildroot je nástroj pro automatické generování Linuxového systému pro malá vestavěná zařízení. Umožňuje vytvořit kompletní systém pro různé procesorové architektury. Systémem je myšleno Linuxové jádro, kořenový souborový systém a obraz disku pro bootloader [13]. Je schopný stáhnout a automaticky zkompilovat celou řadu programů.

Buildroot používá ke své konfiguraci linuxový Kconfig jazyk a jeho nástroje. Zbytek projektu je implementován výhradně za pomocí předpisů pro program `make`, takzvaných makefile souborů.

2.5 SciPy

SciPy je sada balíčků pro jazyk Python určený k matematickým výpočtům. Užívá se pro vědecké účely. Skládá se z několika dílčích projektů [14].

Svými vlastnostmi je podobný aplikacím jako Matlab, SciLab a GNU Octave. Ale zatímco ty používají vzájemně podobný způsob zadávání příkazů ve vlastním formátu, SciPy je postaveno nad programovacím jazykem Python. Stejně jako ostatní zároveň umožňuje práci například se symbolickými proměnnými, n-dimenzionálními poli a vykreslování výsledků a grafů.

Pro tuto práci je stěžejní knihovna NumPy. A především její implementace operací lineární algebry, ze které bude využita funkce pro metody nejmenších čtverců [15].

3 Související práce

S touto prací souvisí různé další projekty, nejenom ty popsané v sekci 2. A to proto, že buďto byly inspirací při implementaci, jako v případě Undertakeru, nebo protože řeší podobný problém.

3.1 Undertaker

Undertaker je program provádějící analýzu Linuxové konfigurace a zdrojových kódů. Kontroluje strukturu preprocesorových direktiv vůči rozličným konfiguracím [16]. Jeho cílem je nalezení tzv. mrtvého kódu, tedy kódu, který nelze žádnou konfigurací aktivovat. Také odhaluje případné bloky, které mají závislosti a které jsou v určitých případech vždy pravdivé. Upozorňuje tak na možné nadbytečné konfigurační volby.

Součástí nástroje je preprocesor, který prochází zdrojový kód a extrahuje z něho závislosti jednotlivých bloků kódu [17]. Takto jsou získány jejich závislosti na ostatních konfiguračních volbách z pohledu zdrojového kódu. Další částí nástroje je generátor konfigurací. Ten načítá závislosti samotných konfiguračních voleb. Dochází ke kontrole takto generovaných závislostí vůči závislostem jednotlivých bloků kódu [18].

S touto prací Undertaker souvisí především díky tomu, že sám již implementuje podobné akce s konfiguračními volbami jako vyžaduje tato práce. Rozdílem ale je, že provádí analýzu bez překladu a spouštění Linuxu. Odhaluje tím tak konfigurační problémy.

3.2 Balíčkovací systémy

Moderní linuxové distribuce jsou postaveny kolem takzvaných balíčkovacích systémů. Je to technologie, která umožňuje instalovat software z většinou sdíleného úložiště. Prakticky každý software závisí na knihovnách, dalších programech, či jiných speciálních souborech, tedy na dalších balíčcích. Balíčkovací systém musí zajistit, aby nainstalovaný software měl splněné všechny závislosti, i když původně v systému nemusely vůbec být. Vzniká tak rozhodovací problém, který musí určit, které balíčky je třeba také nainstalovat, a případně které je třeba odstranit. Pro hledání řešení platí pravidla, že po provedení dané akce (instalace/odstranění balíčku) musí být splněny závislosti všech nainstalovaných balíčků. Což může vést někdy k nepředpokládanému chování [19]. Různé distribuce řeší tento problém různými nástroji. Ukázkou může být právě projekt libzypp [19].

4 Architektura řešení

Cílem je implementace nástrojů, které budou umožňovat generování různých konfigurací Linuxu. Ty poté budou spouštěny a ohodnocovány pomocí výkonových testů. Výsledky (naměřené hodnoty) budou analyzovány. Předpokladem je, že by z nich mohlo být možné vypočítat, jak která volba přispívá k výsledku testu.

Samotná implementace řešení se dá rozdělit do několika samostatných podúkolů. Naším cílem je procházet přes různé konfigurace Linuxu. Proto potřebujeme nástroj, který načte specifikaci konfiguračních voleb v jazyku Kconfig a program, který z této specifikace bude generovat různé přípustné konfigurace. Pro generování řešení bude využit SAT solver. Je tedy třeba programu, který bude načítat Kconfig a jehož výstupem budou závislosti mezi konfiguračními volbami zapsané ve vstupním formátu pro SAT solver. Takový program může částečně využívat zdrojových kódů z nástrojů pro práci s jazykem Kconfig k načítání Kconfig souborů. Převod do formátu pro PicoSAT bude třeba plně implementovat.

Vygenerovanou konfiguraci je následně třeba zapsat ve formátu shodnou se souborem *.config*. Poté může následovat překlad Linuxu.

Dalším úkolem je nabootování. K němu samotnému bude využit nástroj Novaboot, viz sekce 2.3. Ten umožní spuštění sestaveného jádra jak na různých vzdálených, tak virtuálních strojích. Pro měření výkonu budou využity testy, které byly vyvinuty již dříve. Úkolem bude tedy pouze volat Novaboot a ukládat jeho výstup. Další krok, zpracování výstupu, bude přizpůsoben využitým testům.

Posledním krokem bude nalezení vlivu jednotlivých voleb z naměřených výsledků. Vstupem tedy budou výsledky výkonnostních testů a konfigurace, pro které byly testy spuštěny. Algoritmus by měl s ohledem na chyby měření vypočítat, jaké volby se jakým způsobem odráží na výsledku testů. K tomu je využita funkce metody nejmenších čtverců.

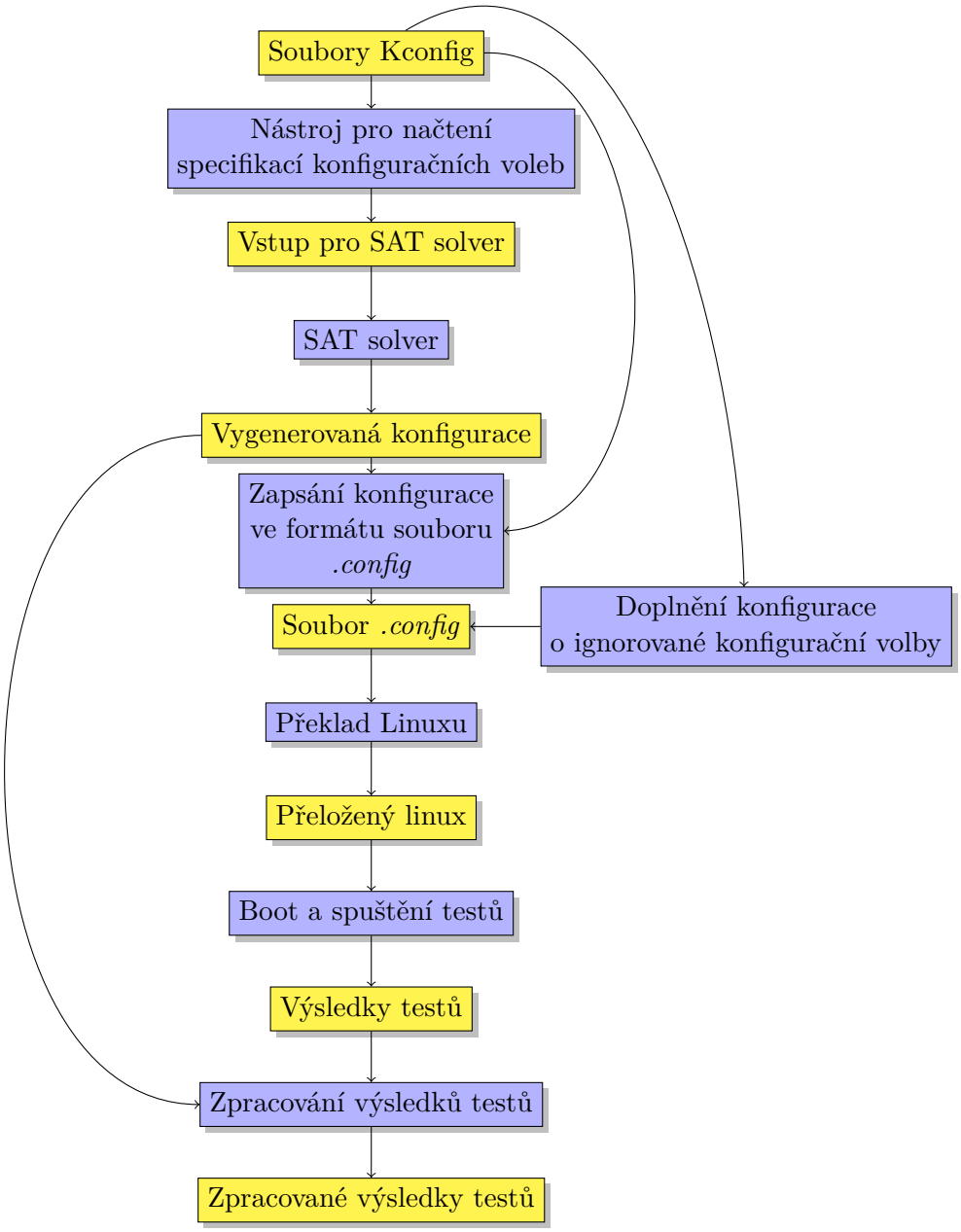
Vzhledem k velkému množství konfiguračních voleb Linuxu jsou výsledné konfigurace velké a je nepraktické je zapisovat k naměřeným výsledkům. Pro zjednodušení tak bude v celém procesu na jednotlivé vygenerované konfigurace odkazováno pomocí jejich hashe [20]. Hash a k ní odpovídající konfigurace bude zapsána zvlášť, což nám umožní propojení výsledků s konfigurací.

Architektura je schématicky zobrazena na obrázku 11.

4.1 Omezení zvoleného řešení

Díky použití SAT solveru je omezením této metody neschopnost pracovat s jinými než booleovskými hodnotami. Pro zjednodušení bude tedy předpokládáno nevyužití modulů v jádře a volby typu řetězec, celé číslo a hexadecimální číslo budou zcela ignorovány.

Zjednodušení z pohledu nevyužití modulu může být pouze dočasné. Omezení na booleovské hodnoty v tomto případě není nepřekonatelné. V této práci to přinese především zjednodušené závislosti a také zmenšený počet možných konfigurací. Velký počet konfigurací bude asi největší problém. Je předpoklad, že počet konfigurací bude mít v nejhorším případě exponenciální závislost na počtu konfiguračních voleb. Přidáním modulů se nám počet konfigurací ještě zvýší.



Obrázek 11 Blokový diagram obecně definující vstupy a výstupy pro navrhnutou architekturu

5 Implementace

Nástroje jsou implementovány ve dvou různých programovacích jazycích. Prvním z nich je C. Jsou v něm napsány všechny samostatné aplikace. Je to z důvodu přímého využití kódu, který je jinak součástí linuxových nástrojů pro práci s jazykem Kconfig a které jsou také implementovány v jazyce C. Zbytek nástroje je implementováván pomocí skriptovacího jazyku Python3. Velká část nástroje je tak rychle a snadno upravitelná. Spouštění nástroje a sestavení programů zajišťuje několik souborů Makefile. Nastavení nástroje zajišťuje jeden soubor *conf.py*.

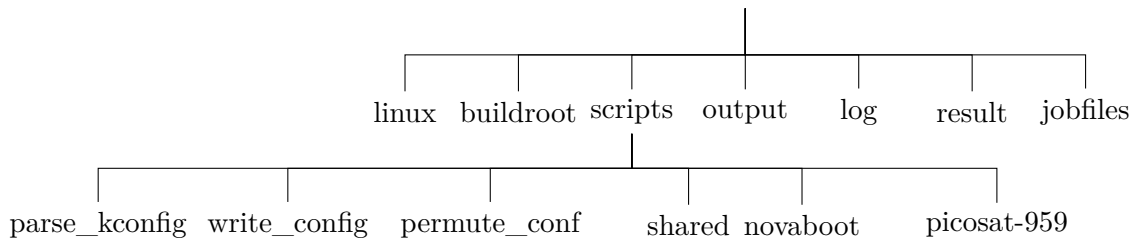
Nejnovější verze nástrojů je vystavena na serveru github.com na adrese: `https://github.com/cynerd/linux-conf-perf`.

5.1 Adresářová struktura implementace

V této sekci je popsána adresářová struktura projektu. V následujícím textu budou názvy souborů uvedeny relativně vzhledem ke kořenovému adresáři projektu.

Složky *linux* a *buildroot* obsahují externí projekty. Další složkou je *scripts*. Ta obsahuje kód pro všechny nástroje, podsložky této složky jsou rozebrány až v sekci o programech 5.2. Výše uvedené složky jsou všechny stálé složky, ostatní jsou vytvářeny dočasně a mazány při čištění projektu. První z takových je *output*. Ta obsahuje výstupy výkonových testů v souborech pojmenovaných hash otiskem konfigurace. Další složkou je *log*. Do té se automaticky ukládají výstupy všech spuštěných programů. Tato složka je zde především pro případ výskytu chyb. Do složky *result* by měl uživatelský skript vytvářet soubory pojmenované podle hashe konfigurace a obsahující hodnotu výkonnostního testu (benchmarku) té konfigurace. Poslední složkou je *jobfiles*. V té se nalézají všechny generované soubory, které nespadají do předcházejících složek.

V kořenovém adresáři implementace se, kromě dokumentačních souborů, nalézá jen *Makefile* a hlavní soubor s nastaveními *conf.py*. Generují se sem ještě jiné soubory, ale vzhledem k jejich různým určením a popisu u odpovídajícího generujícího programu zde nejsou uvedeny.



Obrázek 12 Adresářová struktura implementace

5.2 Programy

Všechny zde popisované programy jsou umístěné samostatně ve své podsložce složky *scripts*. Sdílejí kód umístěný ve složce *scripts/shared*, která obsahuje několik hlavičkových souborů se sdílenými definicemi. Dále se zde nachází složka *kconfig*, ve které se nacházejí zdrojové soubory extrahované z nástrojů Kconfig.

5.2.1 Program parse_kconfig

Cílem tohoto programu je zajistit načtení závislostí ze souborů Kconfig a vygenerovat data pro SAT solver. Tyto data musí být v konjunktivní normální formě, dále jen CNF.

V následujících odstavcích používáme pojem výraz tak, jak byl definován v sekci 2.1.2 o jazyku Kconfig.

Úprava výrazu do CNF

Výrazy načtené z jazyka Kconfig mají obecně jakoukoli formu a je tedy potřeba je převést do CNF. Převod do CNF některých výrazů může vést na exponenciální nárůst počtu termů, například větší počet termů s konjunktivními proměnnými spojené disjunkcemi. Proto je převod proveden tak, že každý term konjunkce/disjunkce proměnných nahradíme jednou proměnnou. Řekněme tedy, že máme dvojici proměnných A a B a zástupnou proměnnou této dvojice bude X. Proměnné A a B jsou buďto konfigurační volby, konstanty nebo zástupné proměnné jiných dvojic výrazů. Můžeme zapsat základní booleovské operace v CNF takto:

$$\begin{aligned} A \wedge B: \quad (X \implies (A \wedge B)) \wedge ((A \wedge B) \implies X) = \\ = X \iff (A \wedge B) = \\ = (X \vee \neg A \vee \neg B) \wedge (\neg X \vee A) \wedge (\neg X \vee B) \end{aligned}$$

$$\begin{aligned} A \vee B: \quad (X \implies (A \vee B)) \wedge ((A \vee B) \implies X) = \\ = X \iff (A \vee B) = \\ = (\neg X \vee A \vee B) \wedge (X \vee \neg A) \wedge (X \vee \neg B) \end{aligned}$$

$$\begin{aligned} \neg A: \quad (X \implies \neg A) \wedge (\neg A \implies X) = \\ = X \iff \neg A = \\ = (A \vee X) \wedge (\neg A \vee \neg X) \end{aligned}$$

Vzhledem k tomu, že jsme se rozhodli nepoužívat moduly (viz. 4.1), jsou při převodu výrazů do CNF všechny výskyty konstanty m nahrazeny konstantou n.

Poslední úpravou je úprava výrazu ekvivalence dvou voleb.

$$A = B: \quad A \iff B = (\neg A \vee B) \wedge (A \vee \neg B)$$

Výraz non-ekvivalence se pak upraví jako negovaná ekvivalence.

S každou konfigurační volbou jsou asociovány čtyři různé výrazy, které odpovídají atributům v jazyce Kconfig. Nahradíme je proměnnou, kterou budeme následně využívat.

- D: Dopředné závislosti, atribut *depends on*.
- S: Zpětné závislosti, atribut *select*, kde uvažovaná konfigurační volba je cílem.

- E: Výchozí hodnota, atribut *default*.
- V: Viditelnost volby, atribut *visible if*.

Pokud není atribut specifikován, má proměnná D hodnotu y a S, E a V hodnotu n. Pro všechny konfigurační volby vygenerujeme následující CNF formuli, kde konfigurační volbu zastupuje A a i když její negace je definována jako další proměnná, je zde uváděna jako $\neg A$:

$$(\neg A \vee D) \wedge (A \vee \neg S) \wedge (A \vee \neg D \vee \neg E \vee V) \wedge (\neg A \vee S \vee E \vee V)$$

Tato formule zajišťuje zvolení konfigurační volby, pokud jsou splněny závislosti. Zároveň vyžaduje splnění dopředných závislostí, pokud je konfigurační volba zvolena. Dále zajišťuje, že pokud není volba viditelná a má splněné závislosti, aby se jí nastavila výchozí hodnota. A pokud není volba viditelná a jsou splněny zpětné závislosti, aby se nastavila hodnota y.

Důležitou pozornost je třeba věnovat výběrům z voleb. Prvním rozdílem od běžných voleb je, že nemusí mít jméno. Jméno je ale užíváno pro propojení jména konfigurační volby a proměnné v CNF. Tento problém je vyřešen generováním jména. Každé volbě, která nemá jméno, je vygenerováno jako *NONAMEGEN* s jedinečnou číslicí na konci. Dalším rozdílem je výlučnost konfiguračních voleb, které pod výběr patří. Tato výlučnost není v načtených závislostech popsána. Musíme ji tak přidat. Vzhledem k tomu, že `parse_kconfig` nepodporuje moduly, není rozdíl mezi výběrem booleovského a třístavového typu. Výlučnost zajišťujeme přidáním jednotlivých vzájemných výlučností. Navíc pokud nemá výběr atribut *optional*, musíme zaručit zvolení alespoň jednoho symbolu v případě, že proměnná představující výběr má hodnotu y. Jak toho docílíme ukáži na ukázce pro tři konfigurační volby A, B a C a výběr H.

$$(\neg H \vee A \vee B \vee C) \wedge (\neg A \vee \neg B) \wedge (\neg A \vee \neg C) \wedge (\neg B \vee \neg C)$$

Nezvolení konfiguračních voleb A, B a C, pokud nemá výběr H hodnotu y, je zajištěna pomocí dopředných závislostí konfiguračních voleb na výběru.

Také CNF formule spojující různé výrazy je jiná. Pro konfigurační volbu $\neg A$ vypadá následovně:

$$(\neg A \vee D) \wedge (\neg A \vee \neg D \vee V) \wedge (A \vee \neg D \vee \neg V)$$

Pokud je uveden atribut *optional*, je poslední závorka vynechána. Získávám tak pouze formuli $(\neg A \vee D) \wedge (\neg A \vee \neg D \vee V)$.

Generování vstupu pro SAT solver

Takto převedené výrazy a vygenerované formule je možné přímo zapsat do souboru *rules* ve formátu DIMACS CNF bez hlavičky, a to nahrazením názvu proměnných zástupnými číslicemi. Z toho důvodu se vytváří další soubor *symbol_map*. Ten obsahuje převod zástupných čísel na jména konfiguračních voleb. Na každé řádce je uvedeno číslo a dvojtečkou od něho je oddělené jméno konfigurační volby. Posledním výstupním souborem je *variable_count*. Ten zaznamenává celkový počet proměnných a je následně využíván pro sestavení hlavičky DIMACS CNF. Na další řádce je ještě zaznamenán celkový počet konfiguračních voleb.

Sestavení a použití

Sestavení programu je prováděno buď z kořenového adresáře projektu pomocí `make parse`, nebo z adresáře *scripts/parse_kconfig* pomocí `make`. Po sestavení získáme spustitelný soubor `parse` ve složce *scripts/parse_kconfig*. Je ho třeba spouštět ze složky

linux. Program požaduje dva argumenty při spuštění. První z nich definuje vstupní Kconfig soubor, většinou je tak tímto parametrem pouze *Kconfig*. Druhý parametr je složka, do které budou uloženy generované soubory. V naší adresářové struktuře tak budeme používat *../jobfiles*.

Program do cílové složky vytvoří soubory *rules*, *symbol_map* a *variable_count*. Soubory jsou popsány v sekci 5.3.1.

5.2.2 Program write_config

Tento program vykonává dvě různé úlohy. Prvním účelem je doplnění chybějících konfiguračních voleb, tedy voleb jiného než booleovského a třístavového typu, to dělá velice jednoduše. Pomocí kódu z Kconfig nástrojů načte Kconfig soubory. Dále načte konfiguraci ze souboru *.config* a po načtení ji jen znovu zapíše. Tím dojde k doplnění chybějících voleb a smazání voleb, které nemají splněné závislosti.

Dalším účelem tohoto programu bylo používání především ve vývoji. Je to implementace, která porovnává vygenerovanou konfiguraci se stavem konfiguračních voleb po načtení. Jedná se o ověření, že všechny závislosti konfiguračních voleb jsou v pořádku načteny programem parse_kconfig (5.2.1). Pokud se odhalí odchylka, je vypsána chyba. Návratová hodnota při ukončení programu odpovídá počtu takových odchylek.

Alternativou k doplnění konfigurace je využití programu `conf` pomocí příkazu `make oldconfig`. To bylo v průběhu vývoje určitý čas využíváno a to pomocí spuštění „`yes "" | make oldconfig`“. Tento program byl původně implementován kvůli kontrolním účelům, příhodně byl využit i k dalšímu účelu. Doplnění i kontrola jsou tak prováděny stejným programem.

Sestavení programu je prováděno buď z kořenového adresáře projektu pomocí `make write`, nebo z adresáře *scripts/write_config* pomocí `make`. Po sestavení získáme spustitelný soubor `write` ve složce programu. Je ho třeba spouštět ve složce *linux*. Při spuštění očekává dva argumenty. Prvním z nich je vstupní Kconfig soubor. Druhý z nich je složka, ve které jsou generované soubory programem parse_kconfig (5.2.1).

5.2.3 Program permute_conf

Tento program interaktivně komunikuje s uživatelem. Umožňuje nastavit, které volby by měly být v průběhu testování neměnné a které proměnlivé. Program čte aktuální konfiguraci ze souboru *.config* ve složce *linux* a při uložení zapíše konfiguraci do souboru *dot_config* do kořenového adresáře projektu. V takto zapsané konfiguraci jsou vynechány konfigurační volby, které jsou označené v programu jako proměnlivé. Konfigurační volby, které jsou neměnné, jsou zapsány i pokud nemají splněné závislosti. Tato konfigurace je následně použita při inicializaci pro generování vyžadovaného řešení skriptem *scripts/initialize.py*, popsaného v sekci 5.3.1. Po změně konfigurace v souboru *.config* je tedy nutné znovu vygenerovat soubor *dot_config* a to načtením a znovu uložením konfigurace pomocí tohoto programu.

Program se ovládá přes textový terminál, kam vypisuje menu obsahující položky typu *bool* nebo *tristate*, bez ohledu na to, jakou mají hodnotu a na splnění jejich závislosti a viditelnosti. Každé položce přiřadí číslo a to je potom využito pro uživatelský vstup. Dále pro každou položku vypisuje znak, který signalizuje, zdali je neměnná nebo proměnlivá. Vypisuje ho mezi značku “<” a “>”. Pokud vypíše “X”, znamená to, že volba je neměnná a všechny volby, které jsou součástí této volby, jsou také neměnné. Pokud je mezi značkami uvedeno písmeno “O”, je volba nastavená jako proměnlivá. Pokud je to znak “-”, znamená to, že volba samotná je neměnná, ale nějaká volba pod touto volbou

je nastavená jako proměnlivá. Pro představu je výstup ukázán na obrázku 13.

```
CAN bus subsystem support
  1<X> Raw CAN Protocol (raw access with CAN-ID filtering)
  2<O> Broadcast Manager CAN Protocol (with content filtering)
  3<X> CAN Gateway/Router (with netlink configuration)
  4<-> CAN Device Drivers ->
Input:
```

Obrázek 13 Ukázka výstupu programu permute_conf pro menu *Networking support/CAN bus subsystem support*

Vstup je řešen vkládáním jednoduchých jedno až dvoupísmenných příkazů, za nimiž někdy následuje číslo. Například nastavení druhé volby jako proměnlivé se provede vstupem „`v 2`“. Možné příkazy jsou rozebrány v následujícím seznamu.

- *e <číslo>* Vstoupí do příslušné volby a zobrazí tak volby spadající pod tuto volbou.
- *u* Vystoupí z volby o jednu úroveň výše.
- *v <číslo>* Nastaví volbu jako proměnlivou.
- *va <číslo>* Nastaví volbu a všechny volby v menu pod ní jako proměnlivé.
- *f <číslo>* Nastaví volbu jako neměnnou.
- *s* Uloží konfiguraci.
- *r* Znovu vypíše seznam konfiguračních voleb v aktuálním menu.
- *h* Vypíše nápovědu.
- *q* Ukončí program.

Program sám o sobě nekontroluje, jestli bude možné proměnlivou volbu v průběhu iterace měnit. Může se stát, že její hodnota bude vynucena nějakou jinou neměnnou volbou se závislostí na tuto volbu, a tak i taková volba se stane neměnná. Toto je problém a do budoucna je nutné kontrolu a hlavně řešení takovýchto situací do tohoto programu implementovat. Další vhodnou možností by také byla možnost jiného než ručního nastavení měřených voleb, například na základě automatické analýzy zdrojových kódů.

Sestavení programu je prováděno buď z kořenového adresáře projektu pomocí `make permute_conf`, nebo z adresáře *scripts/permute_conf* pomocí `make`. Po sestavení získáme spustitelný soubor `permute` ve složce programu. Stejně jako u předcházejících programů ho je třeba spouštět ve složce *linux*. Při spuštění přijímá jediný argument. Je jím vstupní Kconfig soubor. Program je také možné spustit z kořenového adresáře projektu pomocí `make mpermute_conf`.

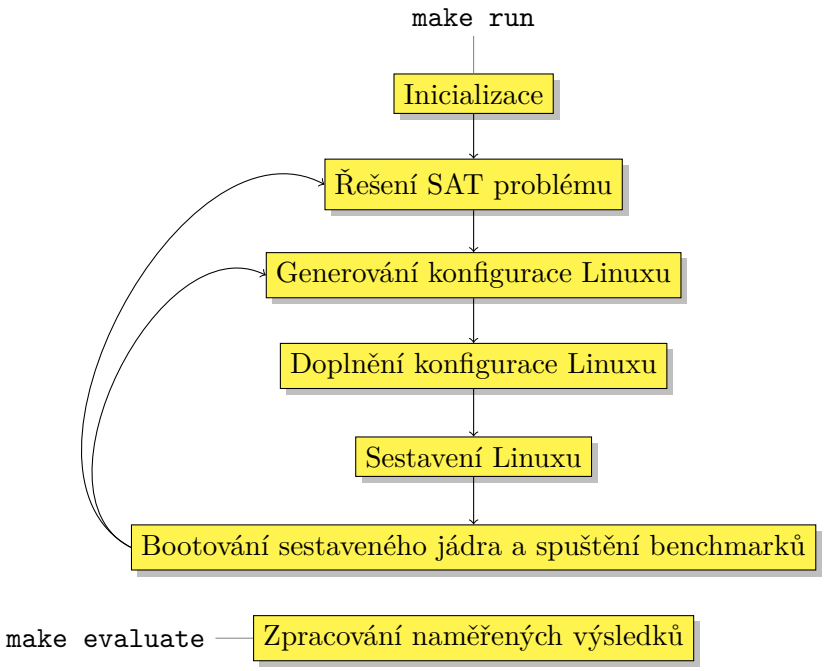
5.3 Kroky generování dat

Samotný proces je rozdělen na kroky. Každý krok generuje nebo nějak upravuje soubory pro následující krok. Jednotlivé kroky tak navazují a na sobě závisí. Kroky jsou:

- Inicializace
- Řešení SAT problému
- Generování konfigurace Linuxu
- Doplnění konfigurace Linuxu
- Sestavení Linuxu

- Bootování sestaveného jádra a spuštění benchmarků
- Zpracování naměřených výsledků

Jednotlivé kroky jsou také zobrazeny na obrázku 14. Obrázek také zobrazuje, jak jsou jednotlivé kroky zapojeny ve smyčce. Skok zpět ze „Zpracování naměřených výsledků“ je podmíněn nastavením nástroje. Podle toho jak je nástroj nastaven, přechází se na krok „Řešení SAT problému“ nebo „Generování konfigurace Linuxu“.



Obrázek 14 Graf zobrazující jednotlivé kroky generování dat vertikálně v jejich návaznosti. Šipky zobrazují smyčku.

Rozdělení do kroků umožňuje zastavení průběhu a následné pokačování. Stejně tak když dojde k chybě, dojde k zastavení na chybovém kroku a uživatel může provést nápravu. Jedná se také o jednoduchý způsob pro testování. Je možné přeskočit některé kroky.

5.3.1 Inicializace

Součástí inicializace je vytvoření složky *jobfiles* a v ní několik souborů. Soubor *iteration* v této složce bude zaznamenávat kolikrát proběhly všechny fáze. Při inicializaci je nastaven na hodnotu 0. Další soubor *phase* obsahuje v textové podobě, která fáze právě probíhá.

Dalším krokem inicializace je spuštění programu parse_kconfig ve složce *linux* s parametry `Kconfig` a *../jobfiles*. Ten vygeneruje do složku *jobfiles* soubory *rules*, *symbol_map* a *variable_count*.

S novými soubory je možné vygenerovat soubor s požadovanými hodnotami neměnných konfiguračních symbolů pro PicoSAT. Vstupem do algoritmu je soubor *dot_config* v kořenové složce projektu. Jména konfiguračních voleb jsou změněna na čísla podle souboru *symbol_map* ze složky *jobfiles*. Čísla jsou záporná, pokud je hodnota konfigurační volby *n* a kladné, pokud je hodnota *y*. Výsledek je uložen do souboru *rules* ve složce *jobfiles*. Zároveň je soubor *dot_config* zkopírován do složky *jobfiles* pod jménem *dot_config_back* za účelem zachování výchozího souboru zpracovávané konfigurace.

Posledním, volitelným, krokem je generování všech validních konfigurací. Generování je popsáno v následující sekci (5.3.2).

5.3.2 Řešení SAT problému

V této fázi se generuje nové řešení. Do dočasného souboru, jakožto vstupu pro SAT solver, se vepíše hlavička formátu DIMACS CNF a za ní obsah souborů *rules*, *required* a *solved* (pokud existuje) z adresáře *jobfiles*. Pro hlavičku formátu DIMACS CNF musíme znát počet proměnných a počet formulí. Počet proměnných je možné načíst ze souboru *jobfiles/variable_count*. Počet formulí odpovídá součtu počtu řádků ve vpisovaných souborech *rules*, *required* a *solved*.

SAT solver, v našem případě PicoSAT, se pokusí nalézt řešení. Když ho nalezne, je zapsáno do souboru *jobfiles/config_map* spolu s hodnotou hashe řešení. Negované řešení je také zapsáno jako jedna disjunktní formule do souboru *solved*, což zajišťuje, že při dalším generování řešení již toto řešení nebude vygenerováno.

Pokud se žádné řešení nenalezne, jsou všechna řešení vygenerována. V této situaci dojde k ukončení smyčky kroků.

S každým opakováním je hledání řešení zdlouhavější, protože roste velikost vstupních dat. Alternativou je vygenerovat všechny konfigurace při inicializaci. To může být časově velmi náročné, ale zrychlí se tak následná smyčka kroků. Také generování všech řešení pouze v rámci SAT solveru je úspornější.

5.3.3 Generování konfigurace Linuxu

Cílem této fáze je přepsat řešení nalezené SAT solverem do souboru *.config*. Hashe doposud řešených konfigurací jsou uchovávány v souboru *jobfiles/config_solved*, pokud soubor neexistuje, pak zřejmě nebyly zatím vyřešeny žádné konfigurace. Podle hashe konfigurace se rozpozná, které konfigurace v souboru *jobfiles/config_map* ještě nebyly řešeny a jedna z nich je načtena.

Načtená konfigurace se zapisuje do souboru *linux/.config* podle syntaxe, jak je popsána v sekci 2.1.4. Při zápisu jsou nahrazena zástupná čísla názvy konfiguračních voleb pomocí mapy v souboru *jobfiles/symbol_map*.

5.3.4 Doplnění konfigurace Linuxu

Řešení vygenerované v předchozím kroku není úplné. Stále pozbývá konfiguračních voleb, které nejsou typu *bool* nebo *tristate*. Nástroje pro práci s jazykem Kconfig v Linuxu se při zjištění tohoto problému zastaví a v lepším případě se pouze dotáží uživatele na příslušnou volbu, v horším případě dojde k vyhodnocení celé konfigurace jako chybné a dojde k jejímu nahrazení výchozí. Ani jeden případ ale není vzhledem k požadavku na autonomní běh nástroje možný. Tento problém je řešen pomocí programu write_config, který konfigurace doplní výchozími hodnotami. Je spuštěn ve složce *linux* s parametry `Kconfig` a `../jobfiles`. Zároveň provádí kontrolu konfigurace. Pokud se načtená konfigurace neshoduje s konfigurací, která byla vygenerována, vznikne výjimka a pokud není nastaveno jinak, je smyčka zastavena.

5.3.5 Sestavení Linuxu

Jakmile je zapsána kompletní konfigurace do souboru *.config*, ve složce *linux*, můžeme přistoupit k sestavení Linuxu. Spustí se příkaz `make` ve složce *linux*. Kontroluje se návratová hodnota pro případ, že by sestavení selhalo.

5.3.6 Bootování sestaveného jádra a spuštění benchmarků

Před spuštěním se vytvoří symbolický odkaz pod názvem *linuxImage* ve složce *jobfiles* odkazující na sestavené jádro. Vytváří se proto, že potřebujeme mít sestavené jádro vždy na stejném místě, abychom k němu ze skriptů mohli jednoduše přistupovat. Ale Linux sestavuje jádro do různých složek podle architektury a pojmenovává ho různými jmény podle zvoleného formátu komprese.

Nabootování se provede voláním Novabootu, jemuž je jako vstup vložen soubor *nb-script* ze složky *scripts*. Předpokládá se, že uživatel nastaví novaboot skript a spouštěný benchmark tak, aby naměřená hodnota byla vypsána na sériovou linku a objevila se tedy na výstupu programu Novaboot. Nejjednodušeji toho lze docílit přesměrováním Linuxové konzole právě na sériovou linku pomocí parametru „*console=ttyS0*“ předávaného Linuxu. Celý výstup Novabootu je zapsán do souboru s pojmenovaném pomocí hashe spouštěné konfigurace do složky *output*.

5.3.7 Zpracování naměřených výsledků

Před tím, než bude možné zpracovat naměřené výsledky, je nutné, aby uživatel nějakým jiným programem načetl hodnoty ze souborů ve složce *output* a zapsal je do stejně pojmenovaných souborů ve složce *result*. Předpokládá se, že soubory v této složce obsahují jen číselnou reprezentaci výsledku. Mohou obsahovat i více výsledků, ty pak musí být na samostatných řádcích a jsou následně považovány za dvě rozdílná měření nad stejnými konfiguracemi. Takto připravené hodnoty jsou načteny a jsou k nim vyhledány příslušné konfigurace v souboru *config_map*. Konfigurace s daty se párují pomocí zaznamenané hashe a názvu souboru s daty.

Zapišme konfigurace do matice $\mathbf{A}$ tak, že řádky odpovídají měřeným konfiguracím a sloupce konfiguračním volbám. Povolené hodnoty v matici jsou pouze 1 a 0. Hodnota 1 určuje ze příslušná konfigurační volba má hodnotu *y*. Naopak 0, odpovídá hodnotě *n*. Dále zapíšeme hodnoty naměřených výsledků do vektoru $\mathbf{b}$. Vektor hledaných dílčích příspěvků označme jako $\mathbf{x}$. Při zachování pořadí konfigurací i konfiguračních voleb mezi vektory a maticí a předpokladu lineární závislosti mezi konfiguračními volbami a naměřenými výsledky (zda je tento předpoklad splněn diskutujeme v sekci 6) platí následující rovnost:

$$\mathbf{Ax} = \mathbf{b}$$

Před řešením omezíme šířku matice $\mathbf{A}$ a s ní příslušně velikost vektoru $\mathbf{x}$. Musíme zajistit, aby sloupce byly lineárně nezávislé. Lineární nezávislost řádků je dána vzhledem k formátu vstupních dat, ale sloupce mohou být lineárně závislé. Pokud by byly sloupce lineárně závislé, dávala by metoda nejmenších čtverců vychýlené výsledky. Redukci tedy provedeme tak, že odstraníme všechny sloupce, které se nemění. Tedy, které jsou pouze složeny z jedniček nebo nul. Dále zajistíme, že žádný sloupec není v matici vícekrát. Odstraníme tak konfigurační volby, které jsou na sobě vzájemně závislé a nahradíme je jednou společnou volbou. Posledním krokem je nalezení základní konfigurace. Ta bude obsahovat ve vypočítané hodnotě také příspěvek neměnných částí systému. Nalezneme ji tak, že hledáme sadu konfiguračních voleb, které se přes všechny konfigurace vylučují a vždy alespoň jedna z nich je zvolena. Pokud žádná taková skupina není nalezena, je přidán do matice *A* sloupek jedniček, který ji bude představovat.

Samotné řešení rovnice je provedeno pomocí knihovny SpiPy a volání funkce pro metodu nejmenších čtverců, `numpy.linalg.lstsq`.

Výsledek je vypsán na standardní výstup. Jednotlivé řádky obsahují jméno konfigurační volby oddělené znakem „=“ od hodnot. Hodnot může být více a jsou vypsány

do hranatých závorek oddělených čárkou. Jeden výsledek se také může vztahovat k více konfiguračním volbám. V takovém případě jsou všechny vypsány před znak „=“ a oddělovány mezerou. Kromě jmen konfiguračních voleb se také objeví název *Base*. Ten signalizuje, že skupina konfiguračních voleb nese v sobě příspěvek neměnných částí systému.

Algoritmus je také schopný pracovat s vektorem **b** jako s maticí. Vektor hledaných dílčích příspěvků se tak také stane maticí. To umožňuje zpracovávat více různých měření najednou. Docílí se toho spuštěním více výkonových testů namísto jednoho. Soubory ve složce *result* pak mohou mít více řádků, kde každý řádek představuje hodnotu pro jiný výkonový test.

5.4 Možnosti nastavení

Chování nástroje je možné upravit změnou nastavení. To je uloženo v *conf.py*. Dalším souborem, který bude muset být pravděpodobně pro různé účely upraven, je skript pro Novaboot, *nbscript* ve složce *scripts*.

Soubor *conf.py* ovlivňuje přímo chování nástroje, zatímco změnou souboru *nbscript* docílíme vhodného nastavení bootování vzhledem k testovanému cíli.

5.4.1 conf.py

Tento soubor je ve skutečnosti Python3 skript. Je tedy třeba dodržovat odpovídající syntaxi. Nastavení v něm zastupují různě pojmenované proměnné.

Nastavení v tomto souboru by nemělo být měněno po provedení inicializační fáze (5.3.1). Změna nastavení po inicializaci může mít nepředvídatelné následky.

Soubor je rozdělen na dvě poloviny. V první jsou proměnné, které pravděpodobně uživatel bude chtít změnit. V druhé polovině jsou definované konstanty. Jednotlivé proměnné v první polovině jsou okomentovány a nemělo by tak být obtížné zjistit, k čemu slouží. V tomto dokumentu budou uvedeny pouze nejdůležitější.

ARCH Řetězec v této proměnné určuje s jakou cílovou architekturou bude nástroj pracovat. Výchozí hodnotou je řetězec *'x86'*. Je využíván jako proměnná prostředí pro nástroje Linuxu. Ostatní hodnoty tak jsou shodné s definicí procesorové architektury Linuxu.

gen_all_solution_oninit Pokud je tato booleovská hodnota `True` dojde k vygenerování všech možných konfigurací v inicializační fázi (5.3.1). V opačném případě bude vždy vygenerováno jediné řešení ve fázi generování řešení (5.3.2). Výchozí hodnotou je `True` a jedná se také o doporučenou hodnotu, protože je to výkonově méně náročná varianta a zrychluje provádění smyčky při samotném měření.

boot_command a novaboot_args Seznam v proměnné *boot_command* určí program a jeho argumenty, který bude spuštěn pro nobootování systému a zaznamenání výstupu. Je ho zde možné zcela změnit a tak bootovat vlastní metodou, nebo provést operace těsně před nabootováním. Výchozí hodnotou je “['scripts/novaboot/novaboot', nbscript] + novaboot_args”. Pokud je je proměnná *novaboot_args* použita, je možné do ní specifikovat přídavné argumenty pro Novaboot. Ve výchozím stavu obsahuje argument určující jako cíl virtuální stroj Qemu. Nahrazením tohoto argumentu na jiný dojde ke změně bootovaného cíle. Podporované argumenty je možné nalézt v dokumentaci Novaboot [11].

5.4.2 nbscript

Jedná se o Novaboot skript. Očekává se, že bude uživatelsky měněn a to z důvodu, že různé měřící sestavy budou potřebovat jiné nastavení. Ve výchozím stavu obsahuje pouze řádky zajišťující načtení Linuxu a Buildroot ramdisku. Syntaxe tohoto souboru je popsána v sekci 2.3.

5.5 Inicializace a spouštění

Před spuštěním nástroje je nutné udělat několik úkonů. Před samotným začátkem je vhodné zkontrolovat a případně změnit nastavení (viz. předcházející sekce). Dalším krokem je vytvoření výchozí konfigurace pro Linux. Je možné využít příkaz `make mlinux` v kořenovém adresáři projektu. Jedná se pouze o zjednodušení přístupu k programu `mconf`, který je součástí Linuxu. Pro využití ostatních konfiguračních nástrojů se stačí přesunout do složky *linux* a vytvořit konfiguraci podle vlastních zvyklostí. Důležité je vypnout podporu modulů. Pokud tak nebude učiněno, neprojde kontrola v následující fázi.

S vytvořenou základní konfigurací je nyní možné přejít na nastavení symbolů, jejichž vliv na výkonnost budeme měřit. Program pro toto nastavení je možné spustit pomocí `make mpermute_conf`. Pomocí programu permute_conf (5.2.3) je možné pomocí klávesových zkratek nastavit, které konfigurační volby budou konstantní a přes které budeme iterovat. Uložením konfigurace se vytvoří soubor *dot_config*, ve kterém je úplná konfigurace (včetně voleb, které nemají splněné dopředné závislosti) až na volby, přes které iterujeme.

Dalším krokem je kontrola konfigurace samotné a nastavení prostředí. Spuštěním `make test` dojde k přeložení Linuxu a pokusu o spuštění systému i benchmarku. Toto by mělo být nápomocné pro ověření, že zvolená konfigurace umožní spuštění výkonového testu a že prostředí i cílový hardware jsou nastaveny správně. Pokud se zdá, že vše proběhlo správně a ve výstupu je vidět výstup výkonnostního testu (benchmarku), můžeme přistoupit k dalšímu kroku.

Následují dva možné následující kroky – spuštění hlavní smyčky pomocí `make run`, nebo separátní spuštění inicializace pomocí `make init`. Inicializace je součástí hlavní smyčky, ale její spuštění mimo hlavní smyčku umožňuje dodatečnou kontrolu před samotným spuštěním.

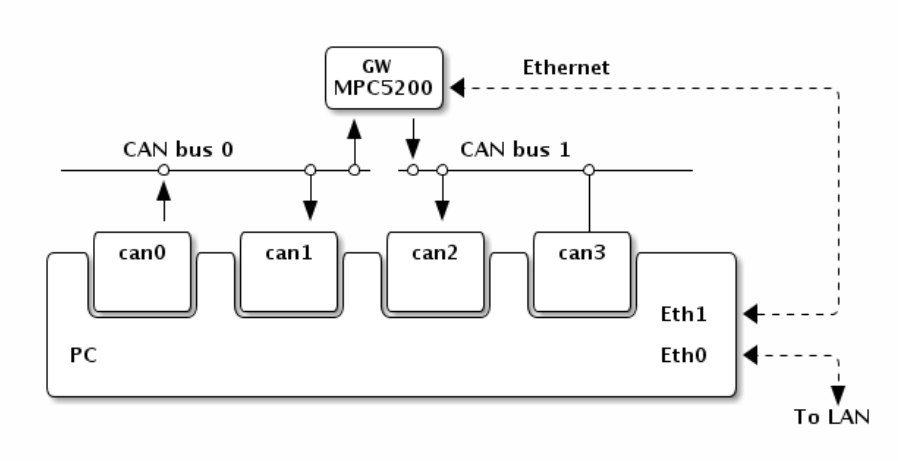
Poté, co doběhla smyčka, je třeba zpracovat výstupy všech výkonostních testů (benchmarků). Výstupy jsou zaznamenány ve složce *output*. Jednotlivé soubory v této složce jsou pojmenovány pomocí hashe vypočtené z dané konfigurace a v souboru je výstup spuštění právě s touto konfigurací. Vzhledem k možné obecnosti je zpracování těchto souborů zcela přenecháno uživateli (příklad zpracování dat je uveden v sekci 6). Následující krok pouze požaduje, aby ve složce *result* v souborech pojmenovaných stejně jako ty v *output*, byly uvedeny pouze čísla. Spuštěním `make evaluete` dojde k vypočtení a výpisu výsledku.

6 Testování a vyhodnocení

Pro ověření funkčnosti a použitelnosti nástroje byl použit na změření malého množství konfiguračních voleb.

6.1 Definice testu

Test bude měřit sběrnici zpoždění zpráv Linuxové brány (gateway) pro sběrnici CAN [21]. Máme řídící počítač a CAN gateway, na které budeme bootovat systém. CAN gateway je vybaven procesorem MPC5200 s architekturou PowerPC. Řídící počítač má PCI kartu s čtyřmi CAN přípojkami. Vzájemně jsou propojeny dvěmi oddělenými CAN sběrnicemi (zobrazeno na obrázku 15). Řídící počítač zasílá rámce na první sběrnici a přijímá je na druhé. Měří tak latenci gatewaye.



Obrázek 15 Propojení počítače a brány k měření [23]

Budeme měřit pro dvě vybrané implementace gatewaye. První z nich je přímo součástí jádra. Druhá využívá funkce `recvmmsg()` a `sendmmsg()` a běží tak v uživatelském prostoru.

Sestavení této testovací sestavy a testů samotných nebylo součástí této práce. Byly pouze využity k naměření dat na praktickém případu. Přesný popis měření je možné nalézt ve článcích odkázaných jako použitá literatura [22, 23].

Budeme testovat Linux verze 4.0.0.

K testu byly vybrány následující konfigurační volby:

CONFIG_SLAB Volí standardní slab alokátor [24]. Volba je součástí výběru v menu *General setup* a výběr má popisek *Choose SLAB allocator*.

CONFIG_SLUB Volí slab alokátor, který minimalizuje použití řádků cache. Volba je součástí stejného výběru jako volba **CONFIG_SLAB**.

CONFIG_SLOB Nahrazuje vnitřní alokátor s drasticky zjednodušenou variantou. Volba je součástí stejného výběru jako volba **CONFIG_SLAB**.

CONFIG_SECURITY_NETWORK Tato volba umožňuje implementaci řízení přístupu k soketům a síti. Volba se nalézá v menu *Security options* a má popisek *Socket and Networking Security Hooks*.

CONFIG_CAN_DEBUG_DEVICES Pokud je volba zvolena, produkují CAN zařízení různé ladící zprávy. Volba se nalézá v menu *Network support/CAN bus subsystem support/CAN Device Drivers* a má popisek *CAN devices debugging messages*.

CONFIG_POSIX_MQUEUE Zapíná POSIX variantu fronty zpráv. Volba se nalézá v menu *General setup* a má popisek *POSIX Message Queues*.

CONFIG_CGROUPS Tato volba přidává podporu slučování skupin procesů. Volba se nalézá v menu *General setup* a má popisek *Control Group support*.

CONFIG_NETFILTER_DEBUG Pokud je tato volba zvolena, produkuje Netfilter ladící výstup zpráv. Volba se nalézá v menu *Networking support/Networking options/Network packet filtering framework* a má popisek *Network packet filtering debugging*.

CONFIG_NET_NS Poskytuje cestu k práci několika procesů se stejným id s různými objekty. Volba se nalézá v menu *General setup* a má popisek *Namespaces support*.

CONFIG_NET_SCHED Zapíná plánovač pro pakety, nahrazuje tak standardní FIFO zásobník. Volba se nalézá v menu *Networking support/Networking options* a má popisek *QoS and/or fair queueing*.

Byly vybrány na základě předpokladu, že by měly ovlivňovat latenci CAN rozhraní. Jedinou výjimkou je konfigurační volba **CONFIG_POSIX_MQUEUE**. Ta byla vybrána jako kontrolní a neměla by měření ovlivňovat. Uvedeme dvě hlavní měření. Jedno pro všech deset konfiguračních voleb. Pro druhé omezíme počet konfigurací pomocí nastavení konfiguračních voleb **CONFIG_SLUB**, **CONFIG_SLAB** a **CONFIG_SLOB** jako neměnné. Kde hodnota *CONFIG_SLUB* je *y*, zbylé dvě konfigurační volby tak mají hodnotu *n*. Bude tak měřeno pouze sedm konfiguračních voleb. Chceme porovnat, jak se výsledná hodnota změní omezením měření.

6.2 Postup nastavení a spuštění

Do složky tohoto projektu byl stažen výše popsaný projekt s výkonovými testy pro CAN rozhraní. Byl vložen do kořenového adresáře projektu a je ve složce *can-benchmark*.

Prvním krokem byla změna cílové architektury. Protože cílové zařízení má architekturu PowerPC, bylo změněno nastavení proměnné **ARCH** na hodnotu „powerpc“.

Dalším krokem bylo modifikace Linuxu pomocí patche *can-benchmark/continuous/steps/shark-ryu-board-patch.patch*.

Poté bylo možné pomocí `make mlinux` vytvořit výchozí konfiguraci. Po jejím vytvoření přišel na řadu na řadu překlad. Pro ten je třeba přidat cestu k překladači do proměnného prostředí. Do proměnné **kernel_env** v souboru nastavení byla cesta přidána pod klíčem *CROSS_COMPILE*.

Pro spuštění testů jsem vytvořil skript *benchmark* v kořenovém adresáři projektu. Jeho obsah je na obrázku 16. Tento skript bude spouštěn namísto standardního příkazu pro bootování (bootování cílového systému je již obsaženo ve volaných skriptech). Byla tak změněna proměnná **boot_command** na „['./benchmark']“.

Testy používají symbolický odkaz na zkompilované jádro *can-benchmark/continuous/gw-setup/uImage*. Pro použití jádra sestaveného nástrojem byl přepsán tento symbolický odkaz tak, aby odkazoval na *jobfiles/linuxImage* relativně ke kořenové složce projektu.

Po těchto úkonech bylo možné provést test. Spuštěním `make test` došlo k překladu Linuxu a spuštění testů.

```
#!/bin/sh
cd can-benchmark/continuous/tests
./kernelgw
./ugw_mmsg_mmsg
```

Obrázek 16 Skript pro spouštění testů

Nyní se mohlo přistoupit k nastavení měřených konfiguračních voleb jako proměnlivé v programu *permute_conf*. Spuštěním *make mpermute_conf* a pomocí interních příkazů byly vybrané konfigurační volby nastaveny na proměnlivé a konfigurace uložena. Poté již bylo možné spustit `make run`.

Po dokončení běhu bylo třeba získat hodnoty z výstupu. K tomu byl použit příkaz: „`for f in $(ls); do grep -o -e '[0-9].* μs $f | sed 's/ μs//' > ../result/$f; done`“., který byl spuštěn ve složce *output*.

Nakonec pro vyhodnocení bylo spuštěně: `make evaluate`.

6.3 Výsledky testů

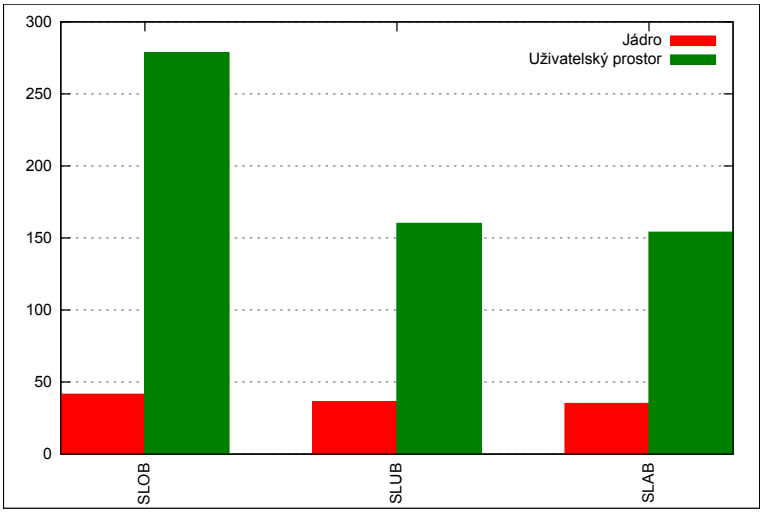
Pro první test bylo vygenerováno a spuštěno 384 konfigurací. Měření trvalo přibližně šest hodin. Pro druhý test bylo vygenerováno 128 konfigurací a měření trvalo necelé dvě hodiny. Výsledné hodnoty pro první měření jsou na obrázku 17 a výsledné hodnoty pro druhé měření jsou na obrázku 20. První hodnota ve výstupu je pro implementaci gateway v jádru. Druhá hodnota je pro implementaci gateway v tzv. user space. Hodnoty mají jednotku μs.

V prvním měření je příspěvek neměnných částí systému zahrnut do třech různých konfiguračních voleb. Mají před sebou napsáno *Base*. Tyto konfigurační volby jsou vzájemně výlučné a zároveň alespoň jedna z nich musela být vždy zvolena. V druhém měření už jsou tyto konfigurační volby neměnné a tak je příspěvek neměnných částí systému zvlášť.

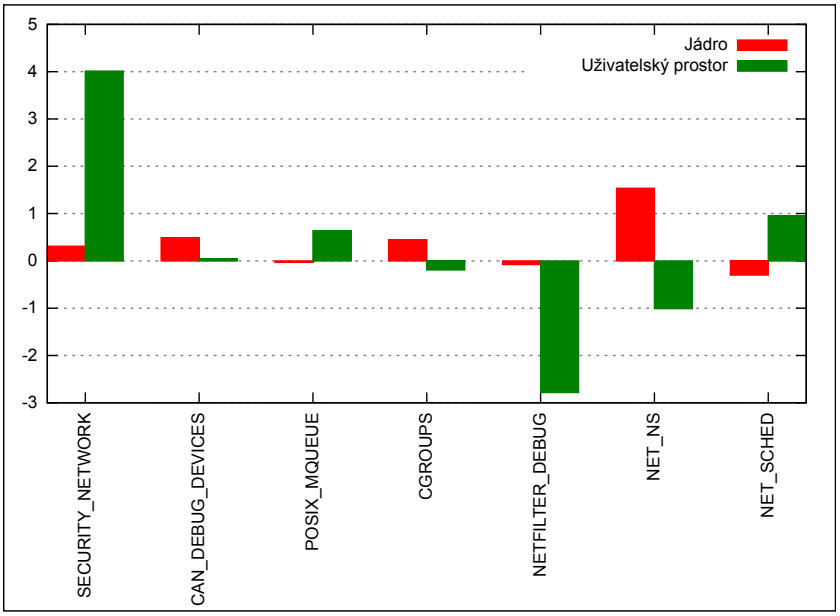
```
Base SLOB = [  41.43352604  278.47021875]
SECURITY_NETWORK = [ 0.3082526   4.01336458]
CAN_DEBUG_DEVICES = [ 0.4871474   0.04340625]
Base SLUB = [  36.11249401  160.06475   ]
POSIX_MQUEUE = [-0.03095156  0.63795833]
CGROUPS = [ 0.44257656 -0.19138542]
NETFILTER_DEBUG = [-0.07601823 -2.78338542]
NET_NS = [ 1.53360885 -1.01079167]
Base SLAB = [  35.08460026  153.98026562]
NET_SCHED = [-0.30071927  0.95195833]
```

Obrázek 17 Výstup nástroje `make evaluate` pro první měření

Zhodnoťme nyní první měření. Je vidět, že změna hodnot mezi konfiguračními volbami `SLAB`, `SLUB` a konfigurační volbou `SLOB` je znatelná, porovnání je v grafu na obrázku 18. Bohužel se nezdá, že by zbylé konfigurační hodnoty měly velkou vypovídající hodnotu. Většina ostatních voleb má totiž hodnotu blízkou nule a tak velmi podob-



Obrázek 18 Graf porovnávající konfigurační volby se zahrnutým příspěvkem neměnných částí systému pro první měření.



Obrázek 19 Graf porovnávající konfigurační volby pro první měření.

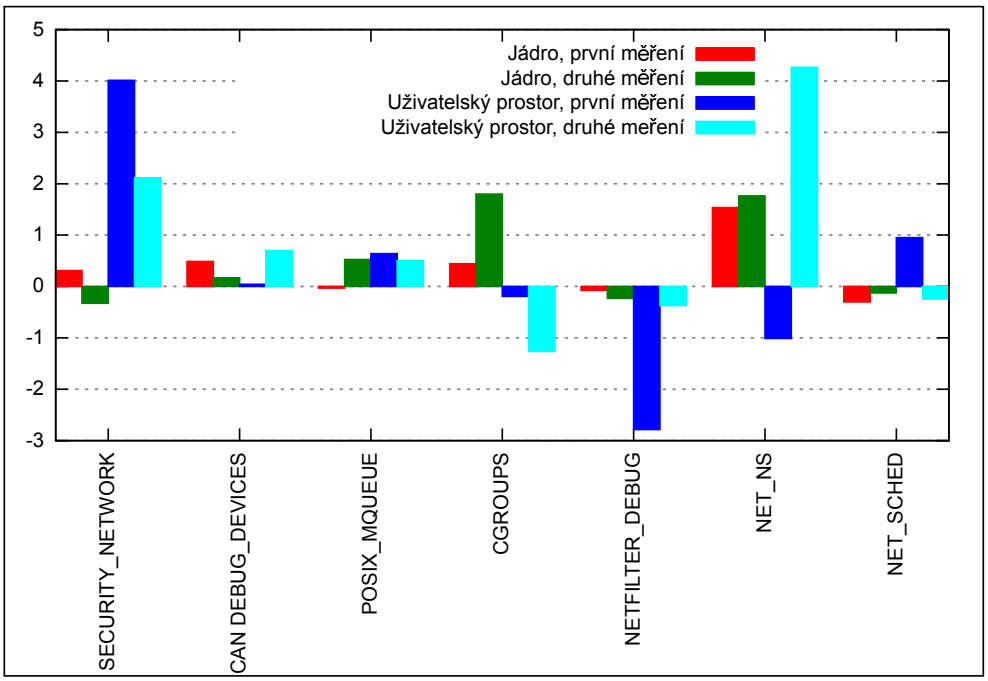
nou volbě `POSIX_MQUEUE` (viz. graf na obrázku 19), která by neměla měření ovlivňovat. Pro jistotu to ověříme. Provedeme měření výchozí konfigurace. Naměřený výsledek je 32.5763 μs pro kernel gateway a 154.341 μs pro gateway v uživatelském prostoru. Když zvolíme konfigurační volbu `POSIX_MQUEUE`, jsou naměřené hodnoty 32.2275 μs a 155.276 μs. Oba výsledky se pro oba testy liší, ale ne výrazně. Považujme to za chybu měření. Pokud bychom tedy prohlásili, že vypočtená hodnota pro konfigurační volbu `POSIX_MQUEUE` odpovídá, mohli bychom prohlásit, že většina ostatních měřených konfiguračních voleb také nemá žádný vliv na výsledný výkon.

Pro další ověření, zdali je výsledek vypovídající, porovnejme ho s druhým měřením.

Řešme opět konfigurační volbu `POSIX_MQUEUE`. Pohledem na druhé měření by se dalo říci, že režie, kterou tato konfigurační volba přidá, je totožná pro oba testy, což odpovídá předpokladu, že nemá na výkonnost vliv. Pohledem na první měření je ale zřejmé, že to

```
SECURITY_NETWORK = [-0.3267125   2.11526563]
CAN_DEBUG_DEVICES = [ 0.16870625  0.69860937]
POSIX_MQUEUE = [ 0.52596875  0.50051563]
CGROUPS = [ 1.79989375 -1.26398437]
NETFILTER_DEBUG = [-0.2295625  -0.36907813]
NET_NS = [ 1.76339375  4.26620313]
NET_SCHED = [-0.12519688 -0.24342188]
Base = [  35.55571563  158.11667187]
```

Obrázek 20 Výstup nástroje `make evaluate` pro druhé měření



Obrázek 21 Graf porovnávající konfigurační volby pro první a druhé měření.

rozhodně není pravda. Pravdou ovšem je, že druhé měření bylo naměřeno především za účelem zjištění, zdali počet konfigurací ovlivní výsledek. Porovnání můžeme pozorovat na obrázku 21. Pokud vyvodíme závěr, že změnou počtu konfigurací dojde k ovlivnění výsledku, je zřejmé, že ani jedno měření není průkazné, protože nemůžeme určit jak byly ovlivněny. Pokud tomu tak není, pak by měly být hodnoty podobné, nikoli odlišné tak, jak jsou.

Před jakýmkoli závěrem se pokusme ověřit, jak hodnoty ovlivňuje chyba měření. Provedeme dvě opakování druhého měření. Získáváme tak třetí měření na obrázku 22 a čtvrté měření na obrázku 23. Zajímá nás, jak se výsledná hodnota bude lišit. Je vidět, že dochází ke znatelným změnám.

Podívejme se ještě jednotlivě na samostatná měření. Použijeme výchozí konfiguraci, ve které vždy pouze zvolíme ručně volbu a spustíme test. V prvním měření byly všechny konfigurační volby nastaveny na „n“. Výsledek je $32.4622\,\mu s$ pro kernel gateway a $154.645\,\mu s$ pro user space gateway. Další měření provedeme se zvolenou konfigurační volbou `SECURITY_NETWORKS`. Naměřený výsledek je $37.6216\,\mu s$ a $160.414\,\mu s$. Konfigurační volbu `SECURITY_NETWORKS` nastavíme na „n“ a provedeme měření se zvolenou

```
SECURITY_NETWORK = [-0.42559688  2.53278125]
CAN_DEBUG_DEVICES = [ 0.24859375  0.8341875 ]
POSIX_MQUEUE = [ 0.53325937  0.50525   ]
CGROUPS = [ 1.83899375 -1.155125  ]
NETFILTER_DEBUG = [-0.20877188  0.59865625]
NET_NS = [ 1.81325937  4.65678125]
NET_SCHED = [-0.18293438  0.58784375]
Base = [  35.50808125  156.82042188]
```

Obrázek 22 Výstup nástroje `make evaluate` pro třetí měření

```
SECURITY_NETWORK = [-0.38453281  2.1705    ]
CAN_DEBUG_DEVICES = [ 0.16545781  0.47459375]
POSIX_MQUEUE = [ 0.61563594  0.93559375]
CGROUPS = [ 1.78713906 -1.01715625]
NETFILTER_DEBUG = [-0.23822656  0.10875   ]
NET_NS = [ 1.81291094  3.767125  ]
NET_SCHED = [-0.20761406  0.10365625]
Base = [  35.56381406  157.54264063]
```

Obrázek 23 Výstup nástroje `make evaluate` pro čtvrté měření

konfigurační volbou `NET_NS`. Naměřil jsem hodnotu $32.1875\,\mu s$ a $166.912\,\mu s$. Provedeme ještě závěrečné měření pro konfiguraci, ve které zvolíme obě konfigurační volby (jak `SECURITY_NETWORKS` tak `NET_NS`). Pro takovou konfiguraci dostáváme hodnoty $40.0247\mu s$ a $167.502\mu s$. Z těchto hodnot je zřejmé, že závislost naměřené hodnoty pro konfigurace se zvolenou a nezvolenou konfigurační volbou není jednoduše lineární pro případ více voleb. Lze tak říci, že výsledky z nástroje, který byl pro tuto práci vytvořen, nejsou průkazné. Je tak zřejmé, že naivní způsob řešení výpočtu, prezentovaný v této práci není dostatečný a možná jiný přístup k vyhodnocování by přinesl výsledky s větší vypovídající hodnotou.

7 Závěr

Práce se zabývala konfigurací překladu Linuxu za účelem analýzy výkonnosti. Domnívali jsme se, že je možné zjišťovat příspěvek jednotlivých konfiguračních voleb na celkovém výkonu jádra, a to pomocí měření různých konfigurací a následnou jednoduchou analýzou získaných dat. Součástí práce bylo vytvoření nástrojů, které umožňují generování všech přípustných konfigurací pro zadané konfigurační volby. Tyto konfigurace jsou následně automaticky spouštěny a je měřen jejich výkon. Pro měření výkonu byly použity existující testy pro komunikační sběrnici CAN. Změřený výkon je zaznamenáván pro další zpracování. Nástroje byly implementovány, ale jak se ukázalo po provedení testů, je zpracování dat, jak bylo navrženo v této práci, nevhodné. Z naměřených výsledků ale nelze prohlásit, že by nástroj byl zcela nefunkční. Věřím, že se jedná o otázku navržení lepšího způsobu zpracování dat, například zkombinováním dat s informacemi ze zdrojových kódů Linuxu a výstupu Ftrace, čímž bychom dostali dodatečnou informaci o tom, zdali se zdrojový kód, přidaný pomocí konfigurační volby, skutečně provede.

Zpracování dat není jediná část, kterou by bylo vhodné vylepšit. S větším počtem proměnných konfiguračních voleb roste velmi rychle počet přípustných konfigurací. S tím také roste čas potřebný pro provedení všech měření. Bude třeba nalézt takové řešení, které zrychlí měření, ale zároveň ho neovlivní.

Literatura

[1] Reinhard Tartler et al. "Automatic OS Kernel TCB Reductionby Leveraging Compile-Time Configurability". In: *Proceedings of the 8th Workshop on Hot Topics in System Dependability (HotDep '12)*. Ed. USENIX. Hollywood, CA, USA, 2012. URL: http://www4.cs.fau.de/Publications/2012/tartler_12_hotdep.pdf.

[2] *Linux kernel source tree kconfig-language.txt*. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/kbuild/kconfig-language.txt (cit. 11. 05. 2015).

[3] *Linux kernel source tree kconfig.txt*. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/kbuild/kconfig.txt (cit. 11. 05. 2015).

[4] *Linux kernel source tree ./init/Kconfig*. URL: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/init/kconfig (cit. 11. 05. 2015).

[5] *DIALOG – Script–driven curses widgets*. URL: http://invisible-island.net/dialog/ (cit. 11. 05. 2015).

[6] *Ncurses*. URL: https://www.gnu.org/software/ncurses/ (cit. 11. 05. 2015).

[7] Wikipedia. *Boolean satisfiability problem — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem (cit. 20. 02. 2015).

[8] Wikipedia. *NP-complete — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/wiki/NP-complete (cit. 20. 02. 2015).

[9] A. Biere. "PicoSAT Essentials". In: *Journal on Satisfiability. Boolean Modeling and Computation (JSAT)* 4 (2008), s. 75–97.

[10] *DIMACS CNF file format*. URL: http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html (cit. 11. 05. 2015).

[11] *Novaboot*. URL: https://github.com/wentasah/novaboot (cit. 11. 05. 2015).

[12] *Novaboot syntaxe*. URL: https://github.com/wentasah/novaboot#novaboot-script-syntax (cit. 11. 05. 2015).

[13] *Buildroot*. URL: http://www.buildroot.org/ (cit. 11. 05. 2015).

[14] *SciPy*. URL: http://www.scipy.org/ (cit. 18. 05. 2015).

[15] Wikipedia. *Least squares — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/wiki/Least_squares (cit. 18. 02. 2015).

[16] *Undertaker*. URL: http://vamos.informatik.uni-erlangen.de/trac/undertaker (cit. 11. 05. 2015).

[17] Christian Dietrich et al. "A Robust Approach for Variability Extraction from the Linux Build System". In: *Proceedings of the 16th International Software Product Line Conference*. Ed. ACM Press. Sv. 1. Salvador - Brazil, 2012, s. 21–30. ISBN: 978-1-4503-1094-9. DOI: 10.1145/2362536.2362544. URL: http://www4.cs.fau.de/Publications/2012/dietrich_12_splc.pdf.

[18] Reinhard Tartler et al. "Configuration coverage in the analysis of large-scale system software". In: *SIGOPS Oper. Syst. Rev. (ACM OSR)* 45.3 (2012), s. 10–14. DOI: 10.1145/2094091.2094095. URL: http://www4.cs.fau.de/Publications/2012/tartler_12_osr.pdf.

[19] *Libzypp satsolver*. URL: https://en.opensuse.org/openSUSE:Libzypp_satsolver (cit. 12. 05. 2015).

[20] Wikipedia. *Hash function — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/wiki/Hash_function (cit. 18. 02. 2015).

[21] Wikipedia. *CAN bus — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/wiki/CAN_bus (cit. 19. 02. 2015).

[22] M. Sojka et al. "Measurement automation and result processing in timing analysis of a Linux-based CAN-to-CAN gateway". In: *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on*. Sv. 2. Zář. 2011, s. 963–968. DOI: 10.1109/IDAACS.2011.6072917.

[23] M. Sojka, P. Pisa a Z. Hanzalek. "Performance evaluation of Linux CAN-related system calls". In: *Factory Communication Systems (WFCS), 2014 10th IEEE Workshop on*. Květ. 2014, s. 1–8. DOI: 10.1109/WFCS.2014.6837608.

[24] Wikipedia. *Slab allocation — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/wiki/Slab_allocation (cit. 20. 02. 2015).

Seznam obrázků