VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

MONITOROVÁNÍ SÍŤOVÉHO PROVOZU S VYUŽITÍM JAZYKA P4

NETWORK TRAFFIC MONITORING USING THE P4 LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **PAVLÍNA PATOVÁ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. TOMÁŠ MARTÍNEK, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Studentka: **Patová Pavlína**
Program: Informační technologie
Název: **Monitorování síťového provozu s využitím jazyka P4**
Network Traffic Monitoring Using the P4 Language
Kategorie: Počítačové sítě

Zadání:

1. Seznamte se s jazykem P4 a dostupnými kompilátory pro převod P4 programů do různých cílových platforem.
2. Seznamte se technologií In-band Network Telemetry (INT) pro monitorování síťových toků.
3. Vytvořte P4 program popisující všechny tři typy monitorovacích uzlů technologie INT (source, transit, sink).
4. Funkčnost vytvořeného programu ověřte s využitím dostupných P4 kompilátorů a případně program upravte dle jejich vlastností a omezení. Alternativně zvažte úpravu samotného kompilátoru.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti dalšího pokračování projektu.

Literatura:

- Dle pokynů vedoucího práce.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Martínek Tomáš, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 29. října 2021

Abstrakt

Dnes se často setkáváme s potřebou monitorovat kvalitu sítě a služeb. K tomuto účelu může posloužit například INT. Cílem je nalézt optimální platformu a s tím spojený překladač pro implementaci INT, pokusíme se tedy k již implementovaným aplikacím (T4P4S, BMv2) nalézt alternativu. Tyto dvě platformy, ale také zmíníme a rozebereme jejich výhody a nevýhody. Výsledkem práce je přehled možností jednotlivých kompilátorů a výkonu popsaných řešení.

Abstract

Today we often encounter the need to monitor network and service quality. For this purpose we can use for example INT. Our goal is to find the optimal platform and associated compiler for implementing INT. We will try to find an alternative to the existing solutions (T4P4S, BMv2). However, we will also mention these two platforms and discuss their advantages and disadvantages. The result of this work is an overview of the capabilities of each compiler and the performance of the described implementations.

Klíčová slova

P4, P4C, T4P4S, SWX, INT, uBPF, eBPF, XDP, BMv2, behaviorální model, Simple Switch

Keywords

P4, P4C, T4P4S, SWX, INT, uBPF, eBPF, XDP, BMv2, behavioral model, Simple Switch

Citace

PATOVÁ, Pavlína. *Monitorování síťového provozu s využitím jazyka P4*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Martínek, Ph.D.

Monitorování síťového provozu s využitím jazyka P4

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing.Tomáše Martínka, Ph.D. Uvedla jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpala.

. . . .
Pavlína Patová
5. května 2022

Poděkování

Chtěla bych poděkovat svému vedoucímu Tomáši Martínkovi za vedení, odborné konzultace a ostatní pomoc při psaní této práce. Dále děkuji svému kolegovi Máriovi Kukovi za pomoc při řešení technických problémů. V neposlední řadě také děkují svým kolegům z FITu a Ondrovi, kteří neváhali sdílet své postřehy.

Obsah

1 Úvod **2**

2 In-band Network Telemetry **3**

3 Jazyk P4 **10**
3.1 Kontrolní a datová vrstva 10
3.2 Struktura P4 programu 12
3.3 P4-14 vs P4-16 16
3.4 Architektonické modely P4 16
3.5 Kontrolní vrstva pro P4 18

4 Překladače pro P4 **20**
4.1 Překladače P4C 20
4.2 Překladač T4P4S 24

5 Implementace jednotlivých uzlů **27**
5.1 Požadavky na implementaci 27
5.2 Implementace pro BMv2 a simple switch 28
5.3 Implementace pro p4c-dpdk 29
5.4 Implementace pro P4C-XDP 33
5.5 Implementace pro P4C-eBPF 33
5.6 Implementace pro P4C-uBPF 33
5.7 Implementace pro T4P4S 34
5.8 Shrnutí 34

6 Testovaní INT aplikace **37**
6.1 Použitý hardware 38
6.2 Použité testovací schéma 38

7 Závěr **46**

Literatura **47**

A INT Metadata **50**

B Spuštění INT pro BMv2 **51**

Kapitola 1

Úvod

Aby síť mohla být efektivně spravována, je zapotřebí provádět monitorování kvality sítě a služeb. K tomuto účelu můžeme využít, například In-band Network Telemetry, jehož speciální vlastností je, že pracuje na úrovni datové vrstvy. Výsledná aplikace díky tomu může operovat rychlostí síťové linky. In-band Network Telemetry je implementovatelná pomocí jazyka P4, což je rozvíjející se jazyk pro programování síťových zařízení. Existuje množství kompilátorů, které překládají P4 program do různých cílových platforem s odlišnými možnosti jak spuštění výsledného programu, tak vůbec použitelnými konstrukcemi, které lze daným překladačem přeložit. Cílem práce je otestovat volně dostupné kompilátory a nalézt vhodný překladač pro implementaci In-band Network Telemetry.

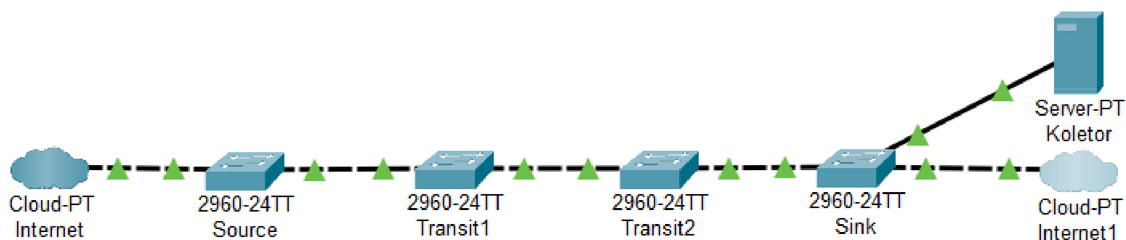
Nejprve se seznámíme s In-band Network Telemetry frameworkem. Ve třetí kapitole bude probrán P4 jazyk pro programování síťových zařízení, historie jeho vzniku a obecně SDN, shrneme si také jednotlivé důležité konstrukce P4 programů a jejich architektury. Ve čtvrté kapitole si více přiblížíme dostupné překladače pro tento jazyk a ukážeme si jejich srovnání. Kapitole číslo pět bude věnována funkcionalitě jednotlivých řešení vytvořených pro různé platformy. V poslední kapitole pak otestujeme vytvořené aplikace z hlediska propustnosti a přesnosti přiřazování časových značek.

Kapitola 2

In-band Network Telemetry

In-band Network Telemetry (INT) je framework navržený pro sběr a monitorování stavu sítě pomocí datové vrstvy. Infrastruktura se skládá ze tří typů uzlů. Zdrojový uzel (angl. source), tranzitní uzel (angl. transit) a koncový uzel (angl. sink), kde *source* je prvním uzlem a začíná zde měření, *transit* je mezi bodem, který sbírá metadata, obecně může být tranzitních uzlů neomezený počet. Posledním uzlem je pak *sink*, který uzavírá měřený úsek sítě [11]. Na obrázku 2.1 je zobrazen příklad infrastruktury se dvěma tranzitními uzly. Některé další pojmy spojené s INT jsou popsány v příloze A.

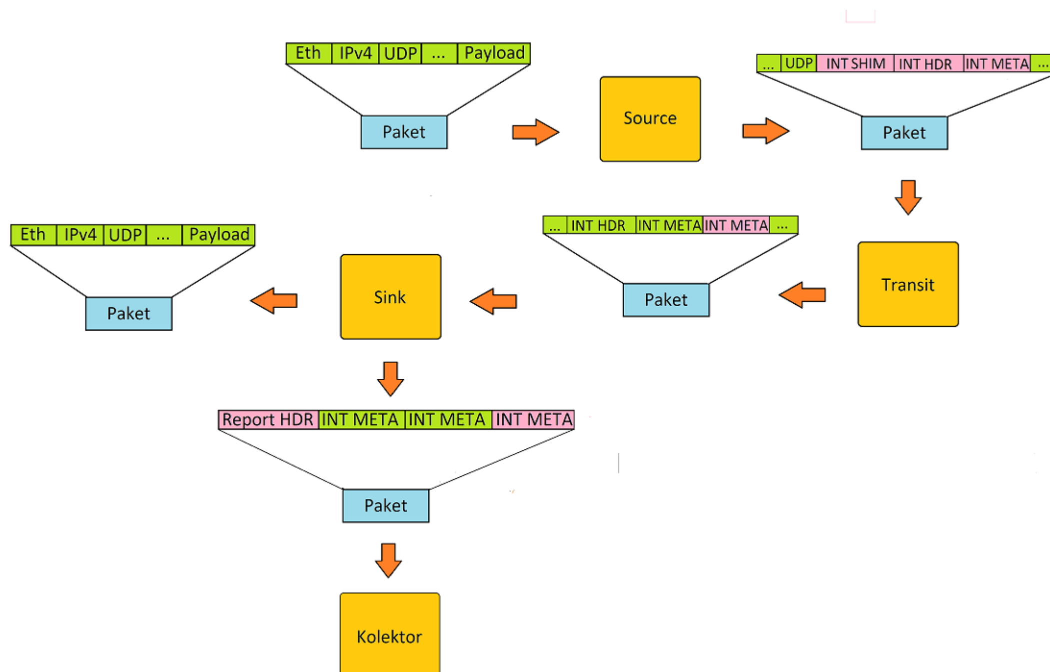
In-band Network Telemetry dovoluje sbírat a hlásit stav sítě přímo pomocí datové vrstvy. V INT architektuře přepínače zpracovávají a přeposílají pakety, které nesou telemetrické instrukce. Pomocí těchto instrukcí je síťovému zařízení sděleno jaké informace má získávat.



Obrázek 2.1: Příklad INT infrastruktury

Existuje několik možností, jak může INT operovat. Prvním typem je INT-MD (eMbed Data). Při tomto operačním módu jsou všechna metadata vkládána do příchozího paketu. Jelikož se metadata a instrukce přidávají přímo do paketu, poměr užitečných přenášených dat se zmenšuje, to lze alespoň trochu omezit maximálním počtem vložitelných měřených metadat případně nastavením maximální možné délky paketu. Stále se nevyhneme zvýšené zátěži na síťová zařízení, která přidávají měřená metadata. Navíc, jelikož se data sbírají

pouze na *sink* uzlu, může docházet ke ztrátě paketů s naměřenými daty [31]. Na obrázku 2.2 můžete vidět příklad INT infrastruktury, která používá právě INT-MD. Pro účely ukázky byl zvolen pouze jeden tranzitní uzel, ale v praxi si jich můžeme za sebe postavit více. Hlavičky přidávané jednotlivými uzly jsou odlišena barevně.



Obrázek 2.2: Průběh INT-MD s jedním tranzitním uzlem

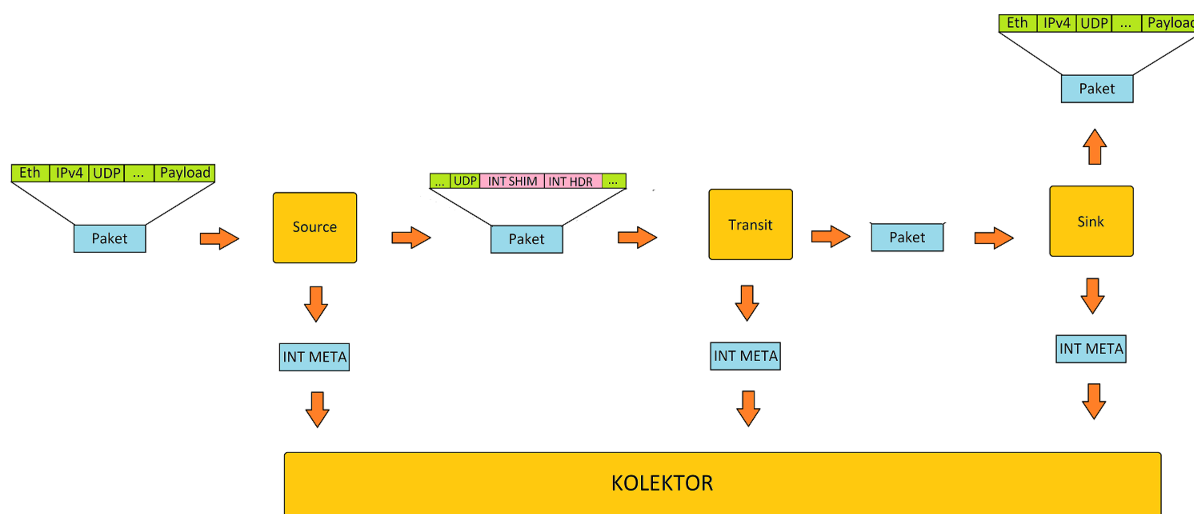
Druhým typem INT je INT-MX (eMbed instruct(X)ions), kdy dochází na zdrojovém uzlu k vložení instrukcí do hlavičky paketu. Ostatní uzly (včetně *source*) pošlou podle těchto instrukcí metadata přímo na kolektor. Koncový uzel poté tyto instrukce odstraní a pošle originální paket ke svému původnímu cíli. Popsaný postup je zobrazen na obrázku 2.3.

Posledním typem je IND-XD (eXport Data). Tento způsob nepřidává žádná metadata. Data jsou posílá podle toho, jak je nakonfigurován jeho Flow Watchlist. Tento mód byl také dříve nazýván *Postcard* [11]. Práce je zaměřena především na první typ INT-MD.

Důležité je i rozhodnout kam INT data včetně hlaviček vložíme. Máme opět několik možností, všechny jsou podrobně popsány v INT dokumentaci, my se zde ovšem budeme zabývat pouze umístěním po UDP hlavičce.

Specifikace INT frameworku se stále vyvíjí, má tedy několik různých verzí. Tato práce se převážně zabývá INT z verze 1.0. První hlavička, kterou budeme vkládat pomocí zdrojového uzlu, se nazývá *INT_SHIM*, její formát je zobrazen na obrázku 2.4. Jak je na něm vidět, skládá se ze tří položek, jež jsou popsány v tabulce 2.1. Druhou hlavičkou je pak *INT_HEADER*. Rozložení a velikost položky můžete vidět na obrázku 2.5. Jejich popis je pak v tabulce 2.2

Za touto hlavičkou už následují metadata z měřených zařízení. Informace o metadatech, které budeme měřit, se zaznamenává do políčka v INT hlavičce zvaného instrukční maska. Skládá se celkem z šestnácti bitů, kde každý označuje specifická metadata. V INT specifikaci jsou ještě interně tyto bity rozděleny na čtyři části všechny po čtyřech bitech. Každá jednotlivá metadata, která připojujeme k paketu, přidávají čtyři nebo osm bajtů k celkové



Obrázek 2.3: Průběh INT-MX s jedním tranzitním uzlem

Název	Popis
Type	Indikuje, který typ INT používáme
Length	Délka, zaznamenává celkovou délku INT hlaviček a metadat jako čtyřbajtové slovo. Slouží hlavně k tomu, aby zařízení, které není INT, mohlo tuto položku přečíst a případně přeskočit přes všechny INT hlavičky k dalšímu obsahu.
DSCP	Slouží pro uchování původního DSCP z IP hlavičky, pokud se tedy toto pole použije pro indikaci INT. V opačném případě, kdy nedochází ke změně DSCP v IP hlavičce, je toto pole rezervováno.

Tabulka 2.1: Popis jednotlivých polí INT SHIM hlavičky

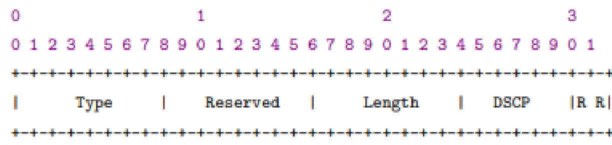
velikosti. Jelikož se instrukční maska po cestě nemění, všechny uzly, přes které INT paket prochází, přidávají obdobná metadata. Tím pádem se s každým projitým *transit* zařízením, rámec zvětší o stejnou hodnotu. Může se ovšem stát, že pakety, jež jsou přijmuty na koncovém uzlu, nemusí mít všechny totožný počet přidaných metadat, jelikož některé nemusely projít identickým počtem tranzitních uzlů [10].

Teoreticky si každý může sám definovat jaká data bude na zařízení sledovat pomocí INT, ale v praxi je užitečné mít definovanou základní sadu metadat pro širší škálu zařízení [10]. Tento set metadat je částečně definován v INT specifikaci. Ale význam některých metadat je popsán pouze stručně, neboť se může lišit v závislosti na zařízení, které data získává. Tabulka 2.3 ukazuje zkrácený popis právě těchto metadat [10].

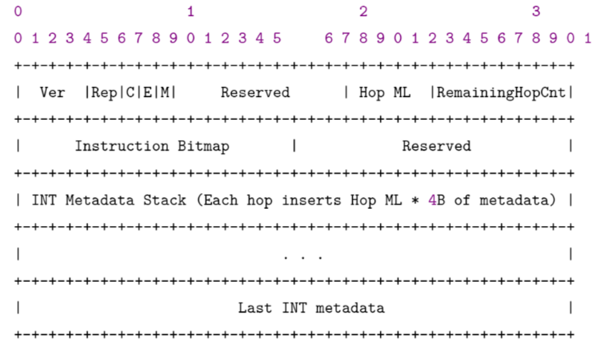
Jak bylo zmíněno, INT pokyny pro přidávání metadat jsou zakódovány jako bitmapa v šestnácti bitovém poli instrukcí přenášeném v INT hlavičce, kde každý bit odpovídá jednomu z metadat popsaných tabulkou 2.3. Většina přidávaných metadat má velikost čtyři bajty, některá jsou složená, jako například zaplněnost front, kdy prvních osm bitů reprezentuje identifikátor fronty a zbylých dvacet čtyři bytů samotnou hodnotu. Jediné pole

Název	Popis
Ver	Ukazuje verzi INT, zde se používá číslo jedna, neboť se jedná o první verzi.
Rep	Požadavek na replikaci, podpora pro tento požadavek nemusí být všude. Pokud toto pole není nula, muže dojít k replikaci paketu.
C	Jednobitové pole C, je použito jako příznak. Pokud dojde k replikaci, koncový uzel musí dokázat poznat klon od originálu. Originál má toto pole vždy nastaveno na jedničku.
E	Druhým jednobitovým políčkem je E, opět se jedná o příznak, tentokrát je nastaven na jedna, pokud zařízení nemůže přiřadit svá metadata k paketu, protože došlo k dosažení maximálního počtu přiřazení.
M	Posledním jednobitovým polem je M, které se nastaví na jedna v případě, že přidáním INT metadat by došlo k překročení nastaveného MTU. V takovém případě zařízení nepřidá žádná metadata a nastaví tento bit na jedna.
Hop ML	Značí délku metadat, přidávaných jednotlivými uzly. Opět je uložena v čtyřbajtových slovech.
Remaining Hop Count	Zbývající počet skoků, označuje, kolik dalších sad metadat může být přidáno INT zařízeními. Na zdrojovém uzlu nastavíme toto pole na takovou hodnotu, aby odpovídala maximálnímu počtu zařízení, které budou přidávat INT metadata do paketu.
Instruction Bitmap	Určuje, která metadata bude každý uzel přidávat. Označuje se i jako instrukční maska.

Tabulka 2.2: Popis jednotlivých polí INT HEADER hlavičky



Obrázek 2.4: INT shim hlavička pro UDP. Převzato z [10]



Obrázek 2.5: INT hlavička pro UDP. Převzato z [10]

ze specifikace, které zabírá o čtyři bajty navíc, má tedy velikost osm bajtů, je identifikátor portu úrovně dva, kdy pro každé ID, vstupní (ingress) i výstupní (egress), vyhrazujeme po čtyřech bajtech. Bity odpovídající jednotlivým metadatům jsou také zobrazeny v tabulce 2.3, kde se bit nula bere jako nejvýznamnější. Jak je také z této tabulky vidět, používá se pouze osm nižších bitů, respektive vyšších, když je nultý bit brán jako nejvýznamnější. Ostatní bity jsou rezervovány a měly by být tím pádem nastaveny na hodnotu nula. Jedinou výjimku tvoří patnáctý bit, který může sloužit pro kontrolní součet bitů v instrukční masce.

Každé zařízení po cestě, které podporuje INT, by mělo přidat svoje vlastní naměřená metadata specifikovaná instrukční maskou za INT hlavičku. Nově naměřené informace budou tedy první před staršími. Jedná se o stejné chování jako při vkládání (pushování) dat do zásobníkové struktury. Toto, možná na první pohled omezení, značně ulehčí následnou implementaci. Nemusíme totiž díky tomu zpracovávat již vložená metadata. K nim se bude program chovat jako k *payload* a přidá je tedy až na úplný konec paketu. Pokud některý uzel není schopen některou hodnotu přidat, měl by na její místo nahrát specifickou hodnotu, tedy samé jedničky, která signalizuje, že příslušné pole není validní.

Zdrojový uzel

Na tomto uzlu začíná měření. *Source* pracuje následujícím způsobem:

1. Ověří, že paket patří do toku, pro který provádíme měření. (INT nemusí být aktivní pro všechny pakety, jež na zařízení přicházejí)
2. Přidáme INT SHIM a INT HEADER hlavičky.
3. Přidáme metadata zařízení.

.

Metadata	Typ	Popis	Bit
Switch id	Zařízení	Unikátní identifikátor přepínače v INT doméně	0
Vstupní port ID	Ingress	Port, na kterém přišel INT paket	1
Vstupní časová značka	Ingress	Lokální čas, kdy byl paket příjat na portu	4
Výstupní port ID	Egress	Port, přes který byl paket odeslán ze zařízení	1
Výstupní časová značka	Egress	Lokální čas, kdy byl paket zpracován egress částí	5
Latence zařízení	Egress	Čas, který paket strávil na INT zařízení	2
Využití výstupní TX linky	Egress	Aktuální vytížení portu, přes který je paket odeslán	7
Zaplněnost front	Egress	Nárůst provozu ve frontě během zpracovávání paketu	3

Tabulka 2.3: Přehled metadat používaných v INT a indikující je bit masky

Tranzitní uzel

Zařízení nakonfigurována do role tranzitního uzlu mají za cíl, přidat svoje metadata k paketu, respektive odeslat je na kolektor, chování závisí na typu INT.

1. Kontrola, že se v paketu nachází INT hlavičky.
   (a) Pokud se vyskytují hlavičky, postupujeme dále.
   (b) V opačném případě je rámec odbaven klasickým způsobem bez získávání či přidávání metadat.
2. Pokud uvažujeme INT typu INT-MD (eMbed Data), který metadata vkládá přímo do paketu a kterému se převážně věnuje tato práce, musí nejprve proběhnout kontrola, zda nebyl překročen maximální počet vložení metadat, případně zda vložení metadat nepřekročí maximální délku paketu. Metadata se musí do paketu vždy vložit jako celek, nelze tedy přidat jen metadata, která by se do požadované velikosti ještě vešla. Pokud se všechna data nevejdou do paketu, nesmí dojít k vložení žádných metadat. To zabezpečí, že paket po jakémkoli počtu tranzitních uzlů bude mít vždy deterministickou velikost, což značně ulehčí práci poslednímu *sink* uzlu.
3. Jestliže jsou všechny výše uvedené podmínky splněny, vloží se metadata do paketu.

Koncový uzel

Posledním z INT uzlů je *sink*:

1. Kontrola, že se v paketu nachází INT hlavičky.
   (a) Pokud se vyskytují hlavičky, postupujeme dále.
   (b) V opačném případě je rámec odbaven klasickým způsobem bez speciální funkcionality *sink* uzlu.
2. Vytvoříme kopii paketu, klon bude sloužit k odeslání výsledků na kolektor.
3. Pro originální paket:
   (a) Odstraníme všechny INT hlavičky včetně metadat.
   (b) Paket pošleme k původnímu cíli.

4. Pro naklonovaný paket:
    (a) Nejprve zkontrolujeme, zda může přidat svá metadata. Jedná se o stejnou kontrolu, která je popsána u funkce tranzitního uzlu.
    (b) Pokud jsou splněny všechny podmínky, můžeme vložit metadata.
    (c) Nezávisle na výsledku kontroly, pošleme všechna doposud získaná metadata na kolektor.

Kapitola 3

Jazyk P4

3.1 Kontrolní a datová vrstva

Běžný postup je nahlížet na vytvářený síťového systém jako na dvě logické části, kontrolní (control plane) a datovou vrstvu (data plane) [9].

Na datové vrstvě probíhají operace s daty, musí být proto uzpůsobena pro jednoduchost a hlavně co největší rychlost provádění akcí. Na druhou stranu kontrolní vrstva, je optimalizovaná pro rozhodování a konfiguraci zpracování na datové vrstvě. Stručně řečeno, kontrolní vrstva představuje řídící logiku, podle které datová pracuje s daty, v našem případě s pakety.

Toto rozdělení přináší řadu dalších výhod:

- umožňuje nám vyvíjet softwarové a hardwarové nástroje pro každou vrstvu zvlášť,
- zlepšení výkonosti, bez kontrolních akcí může datová vrstva pracovat rychleji a může běžet i na specializovaném hardwaru,
- odolnost, pokud nastane problém v kontrolní vrstvě, nemělo by to ovlivnit datovou. I při úplném selhání kontrolní vrstvy, může datová pracovat,
- jednoduchost, software může být aktualizován za běhu, tím dokážeme podporovat kontinuální nasazení, když potřebujeme změnit funkci, pouze vytvoříme instanci datové vrstvy a necháme tudy procházet data bez toho, abychom museli přerušovat provoz celého systému.

Srovnání kontrolní a datové vrstvy je zobrazeno i v tabulce 3.1

Kontrolní vrstva	Datová vrstva
Optimalizováno pro rozhodování	Optimalizováno pro vykonávání
Řízení přístupu	rychlé, vysoká propustnost
Nabízí hodně možností	Jen omezené množství operací
Lehce na-programovatelné	Může být těžší naprogramovat
Podporuje použití cache	Nepodporuje použití cache
Běžně není použito zřetězené zpracování	Může být použito zřetězené zpracování
Zaměřené na dodržování standardu (Policy-driven)	Mechanismem řízené (Mechanism-driven)
Řídí nepředvídatelnost	Předvídatelné

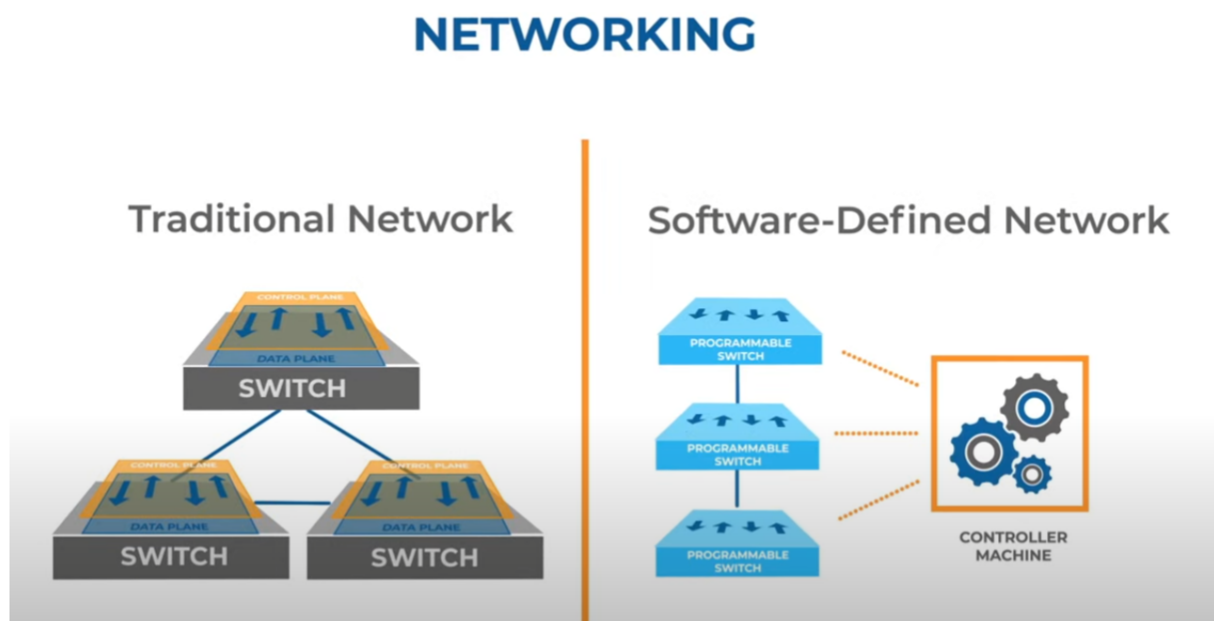
Tabulka 3.1: Srovnání kontrolní a datové vrstvy. Převzato z [9]

Síťové prvky v tradičních sítích mají obě vrstvy spojené, což značně zatěžuje jejich správu, neboť pokud nemáme všechny prvky se stejným konfiguračním rozhraním, není jednoduché takovouto síť udržovat a spravovat. Rozhraní se většinou liší od jednoho výrobce k druhému [14].

Rozdělení kontrolní a datové vrstvy vedlo ke vzniku softwarově definovaným sítím (software-defined networks), nebo-li SDN. Kde datovou vrstvu tvoří zařízení, která posílají pakety k jejich cíly (programovatelné přepínače) a kontrolní vrstvu software, který řídí chování této sítě [9].

SDN & OpenFlow

Jak již bylo zmíněno v předchozí části, chceme mít společný nástroj pro konfiguraci všech zařízení v síti. Toho se právě SDN snaží dosáhnout, přes jednotné aplikační komunikační rozhraní předávat příkazy kontroléru, který následně řídí síťová zařízení. Na obrázku 3.1 je zobrazen rozdíl tradiční a SDN sítě z pohledu datové a kontrolní vrstvy.



Obrázek 3.1: Tradiční vs SDN síť. Převzato z [9]

Jedním z prvních a nejpoužívanějších SDN protokolů je OpenFlow od organizace Open Networking Foundation (ONF). Tento protokol dovoluje konfigurovat zařízení, nejčastěji přepínače, pomocí plnění tabulek, které přiřazují síťovým tokům správné akce. Pokud zařízení neví co dělat s příslušným tokem, zeptá se kontroléru, který mu naplněním tabulky určí akci pro daný typ provozu. Přepínače se tak postupně učí, jak se správně chovat k různým paketům [29].

OpenFlow má omezenou flexibilitu, má totiž pevně dané protokoly, které podporuje. Chceme-li použít nějaký nový protokol, musíme čekat na novou verzi OpenFlow. Taková aktualizace, ale může znamenat výměnu celého zařízení [29].

Toto omezení podnítilo snahu o vytvoření programovatelného zařízení[1]. Následně bylo potřeba vytvořit jazyk, kterým by bylo možné tyto přepínače programovat. Zde přichází na scénu jazyk P4, nebo-li Programming Protocol-independent Packet Processors. Jedná se o poměrně nový vysokoúrovňový jazyk pro programování protokolově nezávislých síťových procesorů. Jeho hlavní předností je možnost volně definovat zpracování paketů na zařízení. Tím přináší další úroveň flexibility.

Jazyk P4 nám dovoluje definovat vlastní hlavičky, postup při získávání a sestavování hlaviček paketu. Samotná logika zařízení je pak ovládaná, podobně jako u OpenFlow, pomocí tabulek, kde si programátor nadefinuje dostupné akce a klíče, podle nichž lze rozlišit toky na základě odlišných parametrů hlaviček. Pomocí kontrolní vrstvy poté můžeme nakonfigurovat chování pro různý provoz. Podrobnějšímu popisu chování a struktuře jazyka se budeme věnovat v následující části práce.

3.2 Struktura P4 programu

První verzí jazyka, $P4_{14}$, byla popsána roku 2014 v SIGCOMM CCR článku s titulem Programming Protocol-Independent Packet Processors[2]. Jedná se o jazyk syntaxí velice podobný jazyku C. P4 je určen především pro programování síťových zařízení. První verze $P4_{14}$ předpokládala použití hlavně pro přepínače. Jelikož jazyk vychází právě z potřeby programovat přepínače, i jeho struktura odpovídá potřebám těchto zařízení. Když přijde paket na klasický přepínač, dojde nejprve k získání hlaviček (parsing), následně jsou provedeny nastavené akce, například změna MAC adresy. Nakonec se paket opět spojí dohromady a odešle ven ze zařízení (deparsing). Právě tímto způsobem je koncipován jazyk P4. Má tři hlavní části. *Parser*, *Match-Action blok* a *Deparser*. V předchozí podkapitole jsme si řekli, že můžeme definovat vlastní hlavičky. Toto lze přirovnat k vytváření struktur v jazyce C. Zde vlastně také vytváříme sadu položek, kde každá má určenou svoji velikost. Hlavičky je potřeba nadefinovat pro všechny protokoly jako je Ethernet, IP, a tak dále. To znamená, že jakoukoli hlavičku chceme použít, musíme ji nejprve definovat. Ale právě toto nám umožňuje vytvářet úplně nové hlavičky, které nejsou pevně definované a běžně používané, například INT. Standardní, často používané hlavičky, můžeme mít uložené v jednom souboru a jen je v případě potřeby importovat. Každá hlavička má kromě námi definovaných polí ještě položku, která určuje jestli je hlavička aktivní/validní. Toto pole obsluhuje *parser*, ale lze i měnit ručně v *Match-Action* části. V příkladu 3.1 můžete vidět ukázku definice hlavičky v jazyce P4 [7].

```
header ethernet_t {
    bit<48> dstAddr;
    bit<48> srcAddr;
    bit<16> etherType;
}
```

Výpis 3.1: Definice ethernet hlavičky. Převzato z [30]

[1]Článek představující programovatelný čip: https://conferences.sigcomm.org/sigcomm/2013/papers/sigcomm/p99.pdf

[2]Článek, jež představil P4 jazyk: https://www.sigcomm.org/sites/default/files/ccr/papers/2014/July/0000000-0000004.pdf

```
    parser MyParser(packet_in packet,
             out headers hdr,
             inout metadata meta,
             inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

Výpis 3.2: Jednoduchý *parser*. Převzato z [30]

Získávání hlaviček paketu

V první části dochází k rozložení dat paketu na jednotlivé hlavičky (*parser*). Navíc lze definovat v jakém pořadí hlavičky očekáváme za sebou. Získávání hlaviček paketu můžeme větvit na základě hodnot v položkách hlavičky, kterou jsme již zpracovali. *Parser* při zpracovávání paketu naplní položky zpracovávané hlavičky a zároveň nastaví, že je hlavička validní.

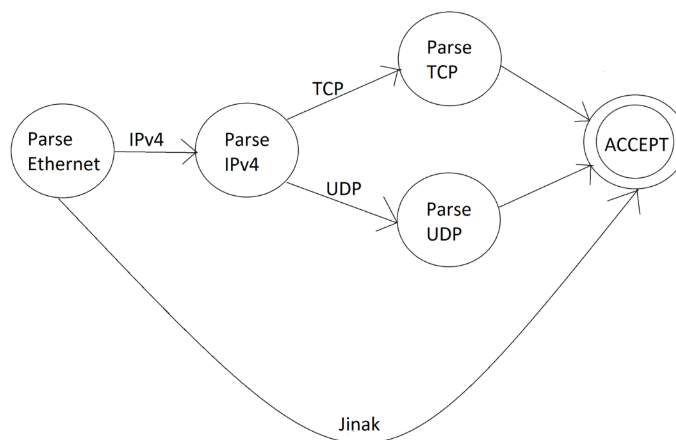
Chování *parser* bloku lze přirovnat k stavovému automatu, kde po každé získané hlavičce následuje další, než dojdeme do některého z koncových stavů. Tento princip je ukázán i na obrázku 3.2. Příklad jednoduchého *parser* bloku je na ukázce 3.2.

Match-action

Hlavní částí programu je *Match-Action blok*, zde definujeme operace, které chceme provádět s paketem. Právě zde se definují tabulky, kterými se ovládá, co bude s pakety provedeno. V *Match-Action* části se vyskytují tři hlavní konstrukce. *Apply* blok, který funguje jako *main* funkce v běžných programech. Akce, jež provádějí požadované operace a tabulky. Ty vybírají, která ze skupiny akcí má být zavolána.

Každá tabulka obsahuje klíč, podle kterého dokážeme rozlišit jednotlivé toky paketů a tím pádem na nich provádět odlišné akce. Existují tři základní typy klíčů:

- Exact - Bity v klíči i v porovnávané položce se musejí rovnat.
- Longest prefix mask (LPM) - Vybere záznam s nejdelší shodou klíče.



Obrázek 3.2: Ukázka principu jednoduchého *parser* bloku

- Ternary - Kromě klíče se zvolí i maska, která určí, které bity se budou porovnávat.

Pokud dostaneme takzvaný *miss*, nebo-li nenajdeme pro daný klíč záznam v *flow* tabulce, zavoláme výchozí akci tabulky. *Flow* tabulka je struktura, která nám udržuje záznamy, jež programu řeknou, jakou akci provést. Můžeme ji plnit buď konstantními záznamy přímo v P4 kódu, nebo častěji přes kontrolní vrstvu. Každá tabulka má automaticky nadefinovanou operaci *NoAction*. Tato akce je zároveň výchozí, pokud není uvedeno jinak. V ukázce 3.3 je vidět i definice akcí a tabulek. Navíc je vidět i změna nastavení výchozí akce.

Sestavování hlaviček paketu

Poslední částí programu je blok pro sestavování hlaviček paketu (*deparser*). Zde dochází k opětovnému složení paketu. Opět si můžeme sami zvolit v jakém pořadí budeme hlavičky skládat za sebe. Data, která jsme neextrahovali v první části pomocí *parser* bloku, jsou brána jako *payload* a jsou v této poslední části vloženy na konec vytvářeného paketu. Paradoxně se více podobá na *Match-Action* část než na *parser*, neboť i zde najdeme *apply* část, kde máme větší prostor pro programování. Můžeme tedy ještě dokončit některé práce s paketem, například spočítání kontrolních součtů. Hlavní funkcí, která má být v *deparser* bloku provedena je *emit*. Ta způsobí to, že pokud je hlavička validní, přidá ji na výstup. Takto vlastně dochází k zmíněnému opětovnému složení výstupního paketu. Tato funkce je také zobrazena v příkladu jednoduchého *deparser* bloku na ukázce 3.4.

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
    }
}
```

Výpis 3.4: Příklad *deparser* bloku. Převzato z [30]

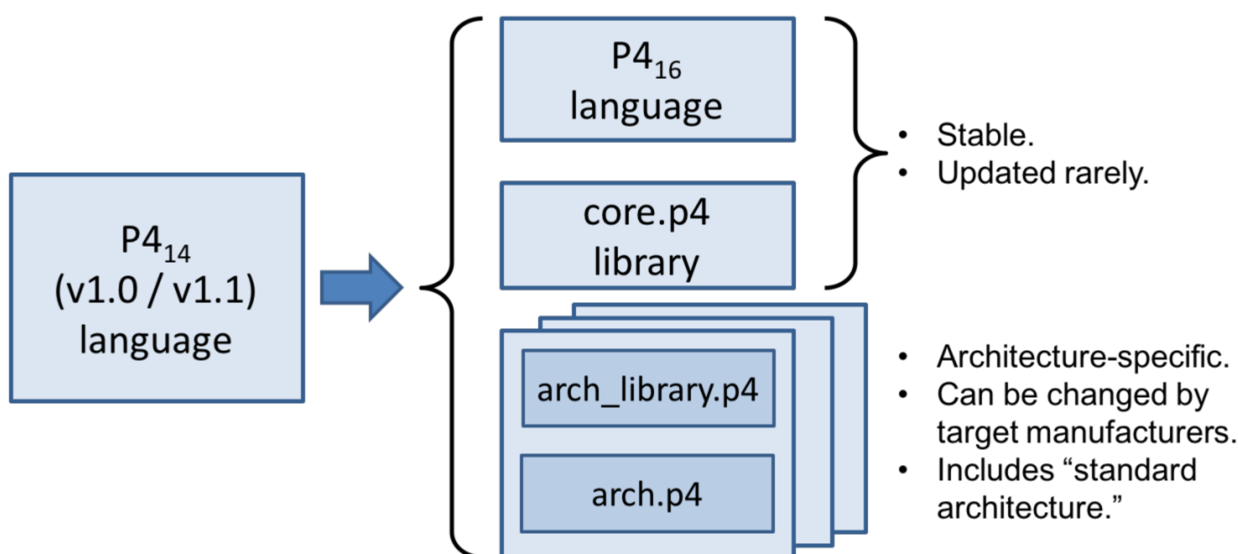
```
control MyIngress(inout headers hdr,
         inout metadata meta,
         inout standard_metadata_t standard_metadata) {

    action drop() {
        mark_to_drop(standard_metadata);
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
    }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = drop();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

Výpis 3.3: Příklad Match-Action bloku. Převzato z [30]

3.3 P4-14 vs P4-16

Jelikož původní P4 jazyk, tedy $P4_{14}$, nahrazoval SDN přepínač, má definován jen jediný model s názvem V1. Model obecně určuje kostru pro P4 program. Budou více rozebrány v pozdější části.

Jak se jazyk P4 začal rozrůstat, vznikaly nové požadavky a kód se stával nepřehledný [7]. Hlavním cílem vydání nové revize jazyka $P4_{16}$, bylo poskytnutí stabilní definice jazyka. Jinými slovy, autoři chtěli zajistit, aby všechny programy psané v $P4_{16}$ zůstaly syntakticky správné a chovaly se stejně pro budoucí revize jazyka. Navíc pokud by některá budoucí verze nedovolovala zpětnou kompatibilitu, měla by být poskytnuta jednoduchá metoda migrace programu na novou verzi [7].



Obrázek 3.3: Evoluce jazyka P4. Převzato z [7]

V porovnání s $P4_{14}$, nový $P4_{16}$ přináší řadu významných zpětně nekompatibilních změn v syntaxi i sémantice jazyka. Při vytváření nové verze jazyka bylo velké množství nativních funkcí z $P4_{14}$ odstraněno a přesunuto do knihoven se základními konstrukcemi pro P4, jak je naznačeno v obrázku 3.3. Díky tomu byl jazyk transformován z komplexního, s více než sedmdesáti klíčovými slovy, do relativně malého základního jazyka s méně než čtyřiceti klíčovými slovy.

3.4 Architektonické modely P4

Možnost výběru architektonického modelu je dostupná až od novější verze $P4_{16}$. Architektonické modely dovolují program v jazyce P4 programovat na různá zařízení ať už hardwarových (NIC, přepínač, směrovač) nebo softwarových (DPDK). Navíc pomáhají izolovat programátora od hardwarových detailů [5]. Architektura identifikuje bloky, které je možno naprogramovat. Většinou se jedná o *ingress* a *egress* kontrolní bloky, *parser* a *deparser*. Tyto bloky se nazývají kontrolní a můžeme je programovat pomocí jazyka P4, dále se v

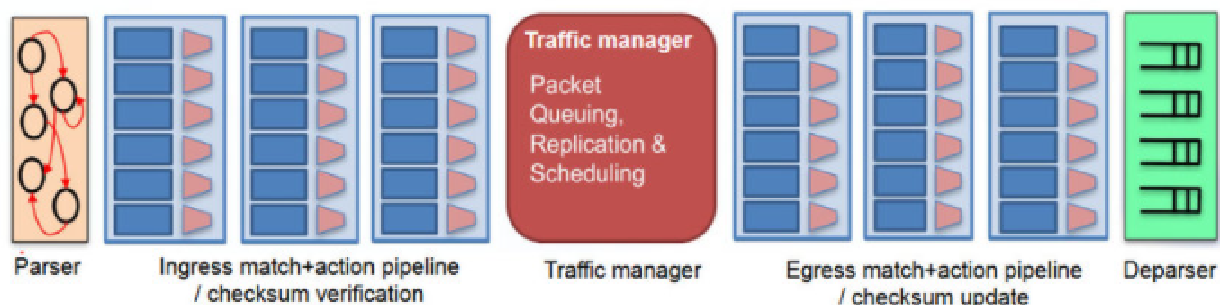
modelu vyskytuje několik fixních bloků, jichž chování je definováno výrobcem. Nelze je konfigurovat přímo pomocí jazyka P4, ale tvůrce musí poskytnout rozhraní pro komunikaci s nimi. Konfigurace těchto bloků se může lišit na každém zařízení [28]. Komunikace P4 programu a zařízení, probíhá za pomoci sady kontrolních registrů nebo signálů. Tyto signály jsou také nazývány vnitřní metadata (angl. intrinsic metadata), jimi může zařízení ovlivňovat program, například informací, z kterého portu paket přichází, nebo i naopak P4 ovlivní zařízení, například při nastavování výstupního portu. Detaily interpretace jednotlivých vnitřních metadat jsou závislé na použité architektuře a mohou být pro každý model jiné.

P4 programy nejsou a nemají být přenositelné mezi architekturami. Naopak programy, které sdílejí stejnou architekturu, by měly být spustitelné na každém zařízení, jež podporují stejný model.

Přímo v P4 specifikaci nejsou definovány standardní modely architektur, nicméně níže si popíšeme dva zástupce modelů.

V1 model

V1 model se výrazně neliší od architektury použité v $P4_{14}$. Díky této podobnosti, lze jednoduše kód napsaný v $P4_{14}$ přeložit do $P4_{16}$ [2]. Původně byl V1 model zaveden jako prozatímní architektura pro překlad $P4_{14}$ dokud nebude definována standardní $P4_{16}$ architektura, tedy PSA model [28][8]. Je implementován v referenčním softwarovém řešení BMv2 Simple Switch, který bude podrobněji popsán v pozdějších kapitolách. Grafické zobrazení V1 modelu můžeme vidět na obrázku 3.4. Lze vidět, že se skládá z *parser* bloku, *deparser* bloku a dvou *Match-Action* částí. Mezi vstupní a výstupní *Match-Action* se nachází blok pro kontrolu provozu (traffic manager), který není programovatelný pomocí jazyka P4.

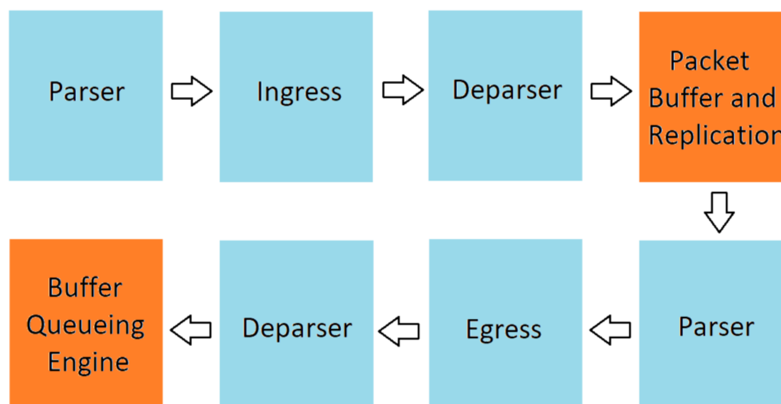


Obrázek 3.4: V1 model. Převzato z [26]

PSA model

PSA (The Portable Switch Architecture) má šest programovatelných P4 bloků a dva bloky s fixní funkcí (angl. fixed-function blocks), fixní bloky jsou na obrázku 3.5 zvýrazněny oranžovou barvou.

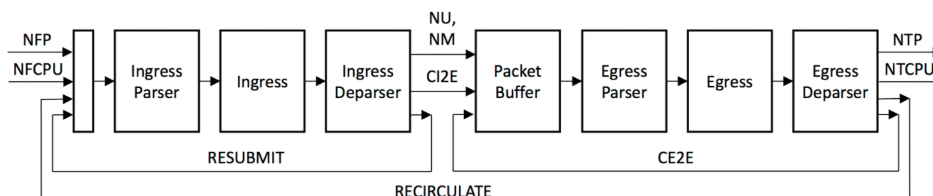
Funkci programovatelných bloků můžeme specifikovat pomocí jazyka P4, způsobem ukázaným v dřívějších částech. PRE (Packet Buffer and Replication Engine) a BQE (Buffer Queueing Engine) jsou vázané na cílové zařízení (target) a mohou být nakonfigurovány pro fixní sadu operací. V PRE lze například nastavit, co se s paketem bude dít po průchodu ingress pipeline. Můžeme provádět různé operace pro replikování provozu tzn. vytvořit více



Obrázek 3.5: PSA model

kopií paketů, které budou poslány na různé výstupní porty a tím získat podporu multicastu. Zpracování v druhé části pipeline probíhá separátně pro každý paket. K replikovaným paketům jsou většinou přidány další metadata, abychom je dokázali rozlišit a mohli na nich provádět další operace. Když použijeme příklad pro klonování, na INT *sink* uzlu by se nám hodilo poslat z výstupního portu dva pakety, kde nám ale na vstup přišel pouze jeden. Použijeme tedy klonování a právě metadata označující replikovaný paket nám pomohou deterministicky rozlišit, o který paket se jedná, abychom neprovedli stejné operace pro oba.

Pakety se mohou dovnitř a ven z obou pipeline dostávat různými cestami. Ty jsou zobrazeny na obrázku 3.6. V této práci se tomuto tématu nebudeme více věnovat, pro detailnější popis všech možností, můžete nahlédnout do specifikace [12].



Obrázek 3.6: Cesty paketů v PSA. Převzato z [12]

Jelikož každá platforma může mít svůj vlastní model, existuje jich více než jen dva zmíněné výše. Jako příklad dalšího modelu můžeme uvést TNA model (Tofino Native Architecture)[3],

3.5 Kontrolní vrstva pro P4

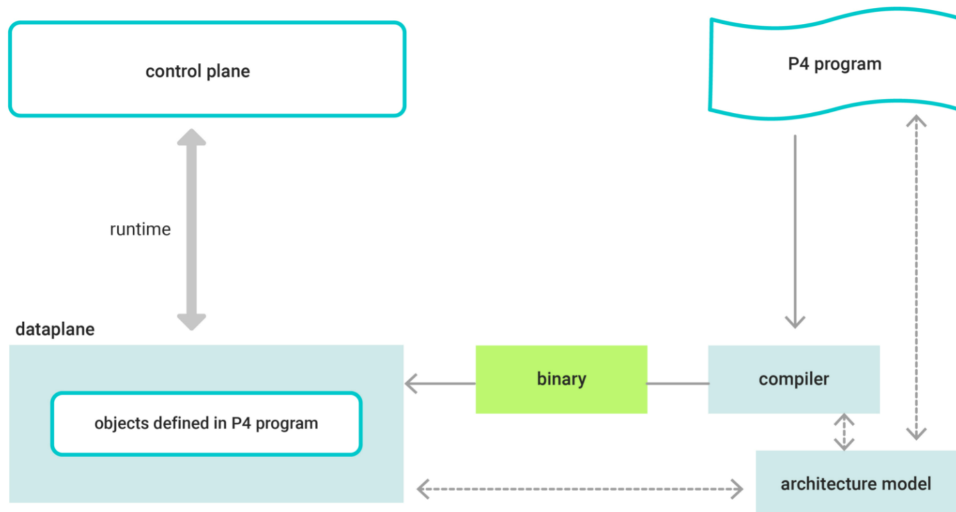
Mnoho cílových zařízení implementuje jak kontrolní, tak i datovou vrstvu. P4 je navržen pouze pro specifikaci datové vrstvy, částečně může i definovat rozhraní, přes které tyto dvě vrstvy komunikují, ale samotný jazyk P4 nemůže být použit pro popis funkce kontrolní

[3] https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf

vrstvy cílového zařízení [7]. Existuje ovšem samostatný projekt P4Runtime, který se zabývá poskytnutím jednotného rozhraní pro konfiguraci P4 zařízení.

Přes kontrolní vrstvu pro P4 zařízení chceme zejména ovládat nastavení tabulek, tím dokážeme využít plnou sílu jazyka a navíc je nám umožněno ovládat jednou nahraný program za běhu. Tedy pro mírně odlišnou funkci zařízení není potřebné jeho restartovat, stačí nám jen pozměnit nastavení tabulek.

Navíc některé funkce nelze bez ovládání přes kontrolní vrstvu vůbec realizovat. Týká se to zejména operací, které zajišťují fixní, tedy neprogramovatelné, bloky.

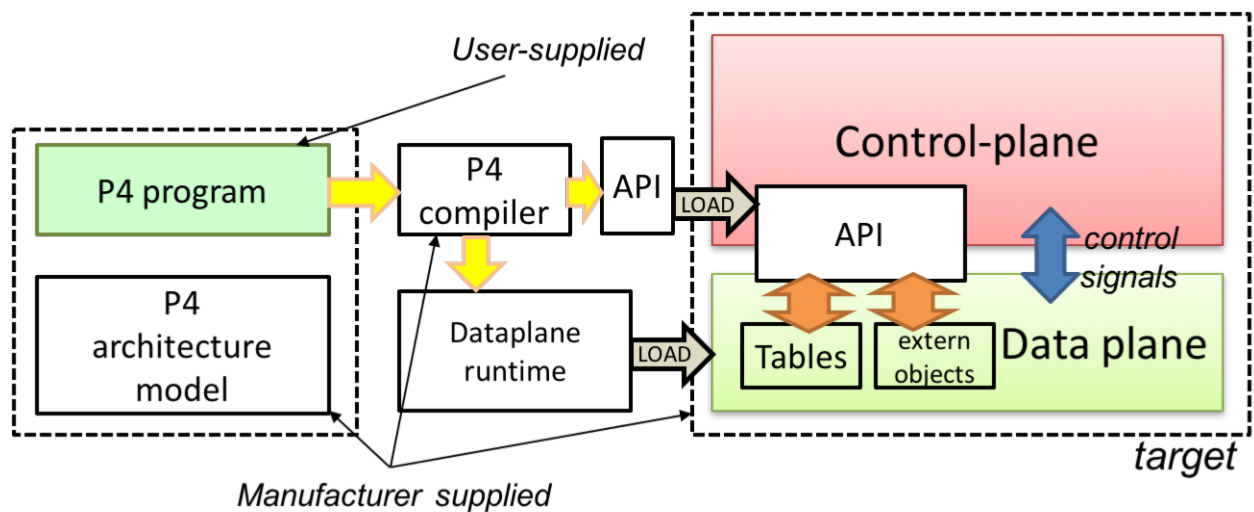


Obrázek 3.7: Kompilace a exekuce P4 kódu. Převzato z [28]

Kapitola 4

Překladače pro P4

Obrázek 4.1 zobrazuje typický pracovní postup při programování zařízení pomocí P4. Můžeme zde vidět, že výrobce zařízení (target) musí nejen dodat P4 překladač, ale i definici architektury. Vlastním přeložením tak získáme konfiguraci pro datovou vrstvu implementující logiku, kterou jsme popsaly pomocí P4 programu [7].



Obrázek 4.1: Typický přístup programování P4 zařízení. Převzato z [7]

4.1 Překladače P4C

P4C je referenční překladač pro jazyk P4. Je modulární a poskytuje standardní frontend a midend, které můžeme zkombinovat s backendem specifickým pro cílové zařízení a vytvořit tak plnohodnotný P4 překladač [15]. To znamená, že pokud chceme vytvořit překladač pro vlastní model, stačí nám implementovat backend a zasadit ho do již implementovaných částí. Některé takto vytvořené překladače si nyní popíšeme.

Překladač P4C-BMv2

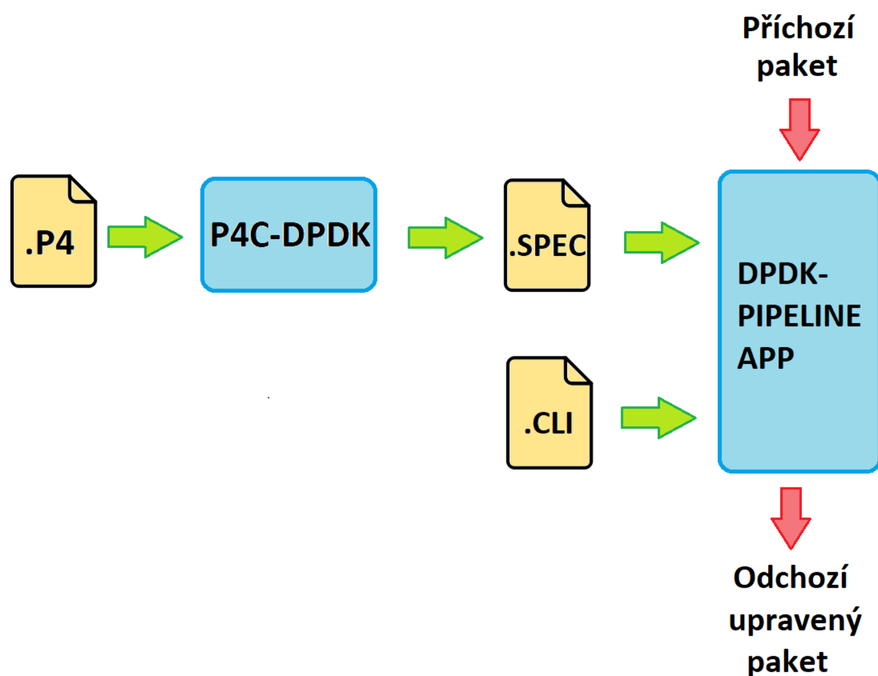
Překládá kód pro dvě architektury. Jednou z nich je behaviorální model PSA přepínače, který používá PSA architekturu. Ale jelikož není ještě zcela dokončen, bude více prostoru věnováno starší architektuře BMv2 pro *Simple Switch* [1].

Ta používá V1 model a dovoluje překládat kód psaný jak ve starším $P4_{14}$, tak i v novějším $P4_{16}$.Výsledkem překladu je pak *json* soubor, kterým můžeme nakonfigurovat náš *Simple Switch*. Samotná aplikace je určena spíše k testování a ladění. Implementace může také figurovat jako referenční řešení problému. Není určen do produkce, neboť se jedná o referenční řešení a je tím pádem výrazně pomalejší než ostatní platformy [4].

Překladač P4C-DPDK

Tento překladač překládá $P4_{16}$ kód psaný pro PSA architekturu[1]. Generuje speciální posloupnost instrukcí, kterými se nakonfiguruje softwarový DPDK přepínač (SWX) [13]. Příklad části takového souboru si můžete prohlédnout na ukázce 4.1. Nejedná se o celý kód, neboť ten je příliš dlouhý. Na ukázce můžeme vidět, že definice tabulky se po vygenerování významně neliší. Program po překladu není rozdělen na jednotlivé programovatelné bloky, ale celá jeho funkcionalita je popsána jedním blokem *apply*.

Na obrázku 4.2 je zobrazen postup konfigurace SWX. Je zde zmíněn i soubor CLI. Ten není nutně potřebný k nastavení interpretu SWX. Není mu proto věnován prostor v této sekci, ale bude podrobněji popsán v kapitole věnující se implementaci.



Obrázek 4.2: P4C-DPDK

K spuštění programu, vygenerovaného P4C-DPDK překladačem, potřebujeme ještě interpret, který dokáže instrukce získané kompilací zpracovat. Tím je již výše zmíněná apli-

[1]Nově i pro PNA, ale zde se tomu nebudeme věnovat, více o PNA architektuře se lze dozvědět na https://p4.org/specs/

```
...

table tb_test {
        key {
                h.ipv4.srcAddr exact
        }
        actions {
                a_test
        }
        default_action a_test args none
        size 0x10000
}
apply {
        rx m.psa_ingress_input_metadata_ingress_port
        mov m.psa_ingress_output_metadata_drop 0x0
        time m.psa_ingress_input_metadata_ingress_timestamp
        extract h.ethernet
        jmpeq MYINGRESSPARSER_PARSE_IPV4 h.ethernet.etherType 0x800
        jmp MYINGRESSPARSER_ACCEPT
        MYINGRESSPARSER_PARSE_IPV4 : extract h.ipv4

...
```

Výpis 4.1: Část kódu vygenerovaného P4C-DPDK

kace SWX (Software Switch pipeline). Jedná se o DPDK aplikaci vytvořenou právě k tomuto účelu. Není ale ještě zcela kompletní, proto nepodporuje všechny potřebné funkce. Vytváří tak po překladači další omezení. Některé konstrukce, které se překladači podaří přeložit, aplikace není schopna zpracovat. Tuto situaci lze sledovat například při použití přetypování, kdy překladači se konstrukci podaří úspěšně přeložit, ale přes interpret se takový kód nedostane.

Překladač P4C-eBPF

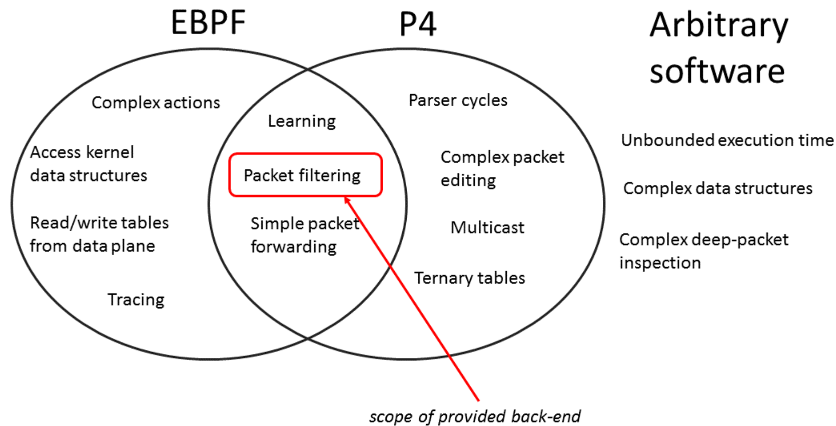
EBPF (extended Berkley Packet Filter) je bezpečný virtuální stroj pro spouštění izolovaných programů v linuxovém jádře. EBPF a jeho rozšíření XDP (eXpress Data Path) efektivně slouží jako programovatelná datová vrstva linuxového jádra. Všechny eBPF programy musí být před nahráním validovány, aby byla zajištěna jejich bezpečnost [23].

Překladač přijímá pouze kód napsaný pro $P4_{16}$ pro speciální eBPF model [18]. Ten se skládá pouze z *parser* bloku a filtru, což je pro implementaci INT nedostačující.

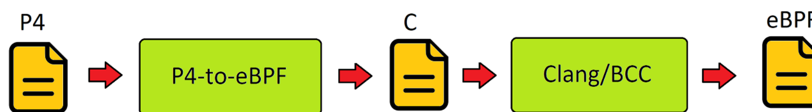
P4 a eBPF mají odlišné silné stránky, ale existují funkcionality, ve kterých se překrývají, zejména v oblasti síťového zpracování paketů [18]. Obrázek 4.3 zobrazuje popisovanou situaci.

Současná verze překladače, překládá kód v $P4_{16}$ do programu v omezené podmnožině jazyka C. Ten je zvolen tak, aby mohl být přeložen do eBPF pomocí clang a bcc[2] [18]. Obrázek 4.4 tento proces ilustruje.

[2]the BPF Compiler Collection – https://github.com/iovisor/bcc



Obrázek 4.3: Přednosti P4 a eBPF. Převzato z [18]



Obrázek 4.4: Překlad P4 do eBPF

Překladač P4C-XDP

XDP (eXpress Data Path) je speciální případ eBPF pro zpracování paketů, který se připojí na nejnižší úroveň síťového zásobníku (network stack). Pakety může prozkoumávat hned jakmile se pakety zkopírují ze síťové karty. Byl původně navržen pro rychlé počítání, zda má být paket zahozen před tím než pro něj kernel alokuje další zdroje, což pomáhá k zabránění DoS (Denial of Service) útokům [23].

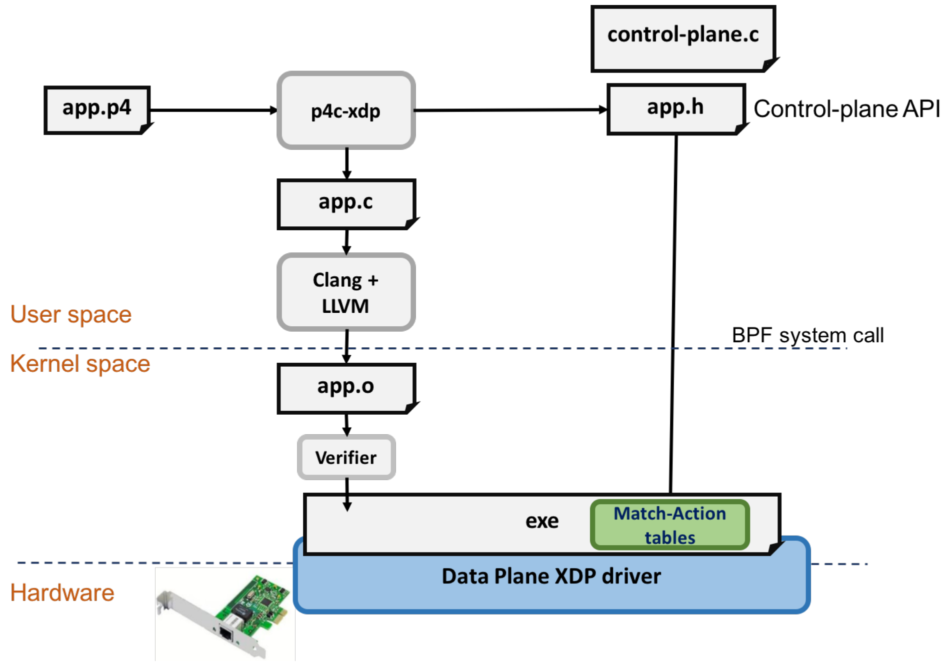
Oproti eBPF překladači toho dokáže více, kromě filtru paketů zvládá fungovat jako jednoduchý přepínač, dokážeme už tedy napsat komplexnější aplikace [23]. Model použitý pro tento překladač se skládá z *parser* bloku, *deparser* bloku a jedné *Match-Action* části.

Ovšem i zde je celkem velké omezení, které znemožňuje napsat komplexnější programy. Má totiž jen omezenou velikost zásobníku (BPF 512 Byte maximum stack size) a složitější aplikace jsou proto zamítnuty při ověřování [21]. Kromě toho nepodporuje broadcast ani multicast, nedokážeme totiž pomocí XDP naklonovat pakety [24].

P4C-XDP překladač také generuje kód v jazyce C, jenž musí být následně přeložen do bajt-kódu [24] a nahrán do přijímací fronty ovladače. Když přijde paket na zařízení, před tím než se přesune do síťového zásobníku, zavolá se uživatelem definovaný XDP program [22]. Obrázek 4.5 ukazuje pracovní postup při použití tohoto překladače. A obrázek 4.6 srovnání jazyka P4 a XDP.

Překladač P4C-uBPF

UBPF virtuální stroj může být použit v každém řešení implementující kernel bypass. Zatímco BPF programy mají běžet v kernelu, uBPF umožňuje BPF programům běžet v *user-space* aplikacích [19]. Stejně jako XDP, podporuje více konstrukcí než eBPF, lze překládat pouze aplikace napsané v $P4_{16}$ pro speciální uBPF model, který stále má jen ingress pipe-



Obrázek 4.5: Pracovní postup při použití XDP a překladače P4C. Převzato z [32]

line [20]. Kromě jednoduchého filtrování paketů můžeme pakety i modifikovat. Naznačení funkce P4C-uBPF překladače můžeme vidět na obrázku 4.7 Vygenerovaný kód lze spustit jen za pomoci P4rt-OVS[3].

Jelikož jsou poslední tři překladače velice podobné uvedeme si jejich srovnání. To můžeme vidět v tabulce 4.1

Vlastnost	P4C-eBPF	P4C-XDP	P4C-uBPF
Implementace	Linux TC subsystem	XDP kernel hook	user-space code
Filtrování paketů	ANO	ANO	ANO
Modifikace paketu a tunelování	NE	ANO	ANO
Jednoduché přeposílání paketů	ANO	ANO	ANO
Registry	NE	NE	ANO
Čítače	ANO	ANO	NE
Kontrolní součty	NE	ANO	ANO

Tabulka 4.1: Srovnání BPF překladačů [25]

4.2 Překladač T4P4S

Tento překladač používá dříve popsaný V1 model. Je multiplatformní jak pro jazyk $P4_{16}$, tak i pro $P4_{14}$. Překládá P4 do C programu nezávislého na cílovém zařízení, který běží nad NetHAL[4]. Hardwarově závislé operace jsou odděleny do NetHAL což zlepšuje přenositelnost,

[3]Programming Protocol-Independent, Runtime Extensions for Open vSwitch using P4: https://github.com/Orange-OpenSource/p4rt-ovs

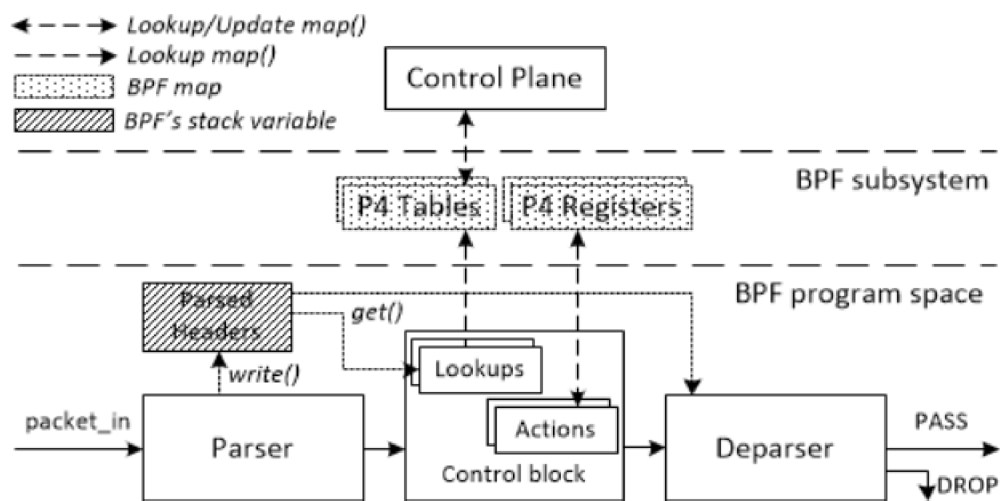
[4]Network Hardware Abstraction Library

P4 vs eBPF/XDP

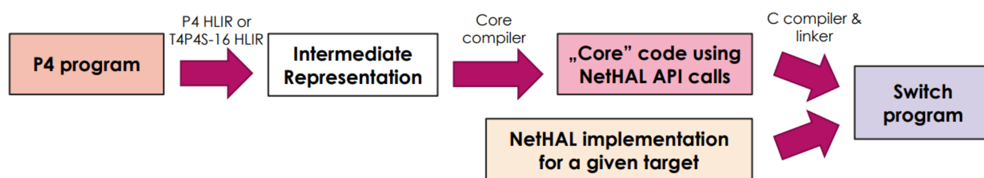
Feature	P4	eBPF/XDP
Level	High	Low
Safe	Yes	Yes
Safety	Type system	Verifier
Loops	In parsers	Tail calls (dynamic limit)
Resources	Statically allocated	Statically allocated
Policies	Tables (match+action)	Maps (tables)
Extern helpers	Target-specific	Hook-specific
Control-plane API	Synthesized by compiler	eBPF maps

Obrázek 4.6: P4 vs eBPF/XDP. Převzato z [24]

to potom ve výsledku znamená, že k podpoře nové architektury musíme implementovat pouze nový NetHAL [17][33]. Na obrázku 4.8 můžeme vidět průběh překladu při použití tohoto kompilátoru.



Obrázek 4.7: Architektura P4C-uBPF. Převzato z [25]



Obrázek 4.8: Postup překladu P4 pomocí překladače T4P4S. Převzato z [16]

Kapitola 5

Implementace jednotlivých uzlů

Našim cílem je nalézt optimální platformu a s tím spojený překladač pro implementaci INT. Nejdříve budou specifikovány obecné požadavky pro aplikaci INT a následně provedeme popis jednotlivých implementací se srovnáním, které požadavky splňují.

5.1 Požadavky na implementaci

Referenční implementace INT, která je popsána přímo ve specifikaci verze 1.0, předpokládá několik konstrukcí, které jsou potřebné, aby bylo možné INT na daném modelu/překladači implementovat v plném rozsahu.

Klonování

Pro plnohodnotnou funkci *sink* uzlu bychom potřebovali mít možnost klonovat pakety. Je to důležité především při používání paketů síťového provozu pro přenos INT dat v síti, kdy se ke standardním datům rámce, přidají naše metadata. Ty musíme na posledním koncovém *sink* uzlu odstranit a originální paket poslat k původnímu cíli. Ale zároveň potřebujeme získaná metadata odeslat na kolektor, aby se uložila a mohla dále analyzovat. Z toho vyplývá, že potřebujeme z koncového uzlu poslat celkem dva pakety, i když nám přišel pouze jeden. A jelikož nám P4 jazyk neumožňuje vytvořit nový paket, musíme si poradit naklonováním příchozího paketu, kdy pro jednu instanci provedeme odstranění metadat a odeslání k původnímu příjemci a druhý rámec pošleme i s metadaty na sběrnou stanici.

Metadata

INT verze 1.0 předpokládá dostupnost několika metadat. Ty jsou z větší části popsány v kapitole 2. INT aplikace by měla všechna tyto metadata umět získat.

Egress pipeline

Tento požadavek velice souvisí s předchozím, neboť některá metadata jsou dostupná až v egress části, přímo se nabízí egress časová značka. Nedostupnost tohoto kontrolního bloku tedy způsobí nedostupnost metadat získávaných v egress bloku.

Časové značky

Časovým značkám je v této práci věnována asi největší pozornost v porovnání s ostatními metadaty, proto je začleněno jako zvláštní kritérium pro srovnání jednotlivých implementací.

Přepočítání kontrolních součtů

Každý uzel přidává (případně odstraňuje) do paketu svoje metadata. Pokud přidáme informaci do paketu, ale nepřepočítáme kontrolní součet (síťové a transportní vrstvy), může dojít k zahození dalšími zařízeními v síti, neboť bude paket vyhodnocen jako chybový. Na konci zpracování paketu pomocí zařízení naprogramovaném pomocí P4 bychom tedy měli přepočítat kontrolní součet. Nelze ho, ale počítat klasickým sčítáním šestnácti bitových polí, nýbrž musíme použít takzvaný subtraktivní kontrolní součet, kdy pole, kde došlo ke změně, odstraníme a následně znovu přidáme. Tento výpočet musíme použít hlavně z důvodu, že jazyk P4 nemá přístup k přenášeným datům paketu (payload).

Ovládání přes kontrolní vrstvu

Posledním požadavkem je, aby bylo možné aplikaci ovládat přes kontrolní vrstvu. Toto se netýká pouze INT aplikace, ale jelikož jedna z hlavních předností P4 a obecně SDN je právě možnost vzdáleného ovládání bez nutnosti přerušení činnosti programu, její nepřítomnost značně omezí možnosti naší aplikace.

5.2 Implementace pro BMv2 a simple switch

Implementace pro behaviorální model (BMv2) Simple Switch vznikla v rámci projektu GEÁNT. Jedná se o jednu aplikaci, ve které je možné spustit jakýkoli uzel (source, transit, sink). Výběr, který uzel bude právě aktivní se při této implementaci provádí přímo přes kontrolní vrstvu.

Splnění požadavků

Tato implementace splňuje většinu požadavků popsaných dříve:

1. Klonování je umožněno, protože jednak překladač správně přeloží požadavek v P4 kódu a zároveň díky kontrolní vrstvě, kterou lze P4 program v celku dobře ovládat.

2. Egress kontrolní blok můžeme použít, díky tomu dokážeme získat převážnou část metadat požadovaných specifikací. Nepodporuje metadata jako Využití Tx egress portu a Port ID úrovně 2.

3. Ve výchozím stavu je do metadat časových značek načten čas od startu aplikace. Toto omezení je odstraněno pokud použijeme upravenou verzi BMv2 přepínače a P4C překladače[1], jež je dostupná pro *docker*.

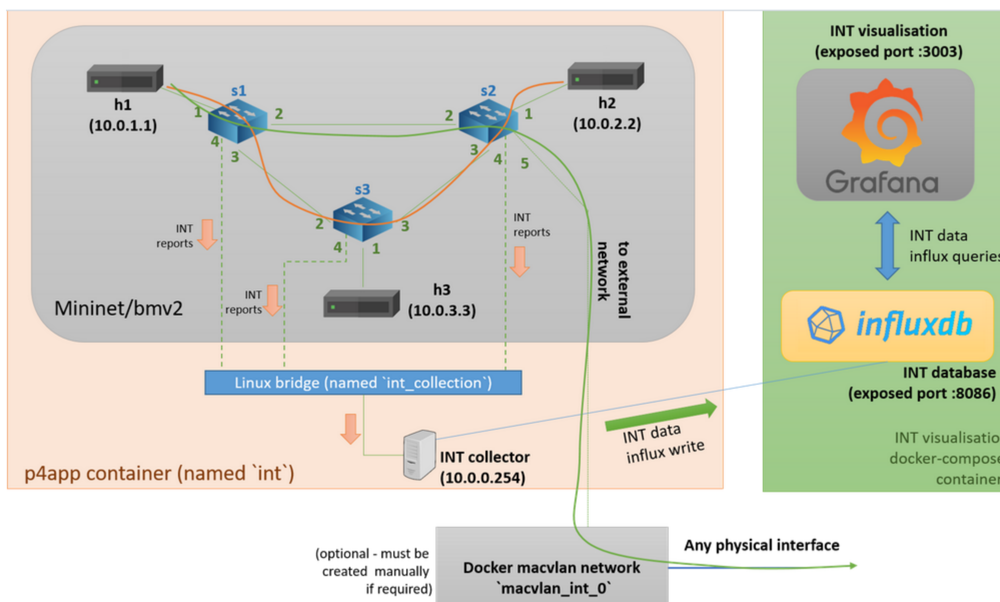
4. Blok pro výpočet kontrolních součtů se vyskytuje přímo v modelu takže může být jejich přepočet bez problémů proveden.

5. Kontrolní vrstva funguje dostatečně pro všechny konstrukce potřebné v INT.

[1] Upravená verze BMv2 pro docker https://github.com/jaxa1337/p4app

Provoz aplikace

Aplikace[2] je primárně spustitelná v emulátoru mininet. Jeho zapojení je zobrazeno na obrázku 5.1.



Obrázek 5.1: Zapojení mininet. Převzato z [27]

Návod jak zprovoznit kód na konvenčním počítači je v příloze B.

5.3 Implementace pro p4c-dpdk

Pro tento překladač neexistuje žádná volně dostupná implementace INT, musela být tedy nově vytvořena. Vycházelo se z referenční implementace popsané ve specifikaci a již implementovaného programu pro překladač BMv2.

Jak bylo popsáno v kapitole o překladači P4C-DPDK, překladem získáme mezikód, který musíme předat SWX aplikaci, jež ho interpretuje a nastaví podle něj svou funkci. K spuštění potřebujeme kromě konfiguračního souboru, jenž získáváme z překladače, takzvaný CLI soubor, ve kterém je popsáno nastavení programu. Pokud ho aplikaci přímo nepředáme, dostaneme se do interaktivního režimu aplikace a můžeme příkazy, který by byly v CLI souboru, zadávat ručně. Tato nastavení nejsou moc zdokumentována, jejich funkci jsem tedy odvozovala především z názvů, popřípadě přiložených příkladů. Právě v tomto souboru je specifikován soubor s instrukcemi pro správné nastavení SWX programu, kromě tohoto nastavení sem můžeme vložit i výchozí nastavení tabulek.

Na ukázce 5.1 si popíšeme strukturu CLI konfiguračního souboru. Jak je vidět na příkladu, máme dva nejdůležitější komponenty, které chceme nastavit:

1. Jednak linka, která může být definovaná buď přímo pomocí PCI-ID (například 0000:86:00.1) nebo pomocí čísla portu jak je zobrazeno v ukázce na řádcích tři a čtyři. Je důležité poznamenat, že v případě definice linky pomocí čísla portu se čísluje od nuly a nejprve se použijí fyzické karty, jež jsou k DPDK připojeny. Mellanox karet se automaticky

[2]GEÁNT implementace: https://github.com/GEANT-DataPlaneProgramming/int-platforms

```
 1 | mempool MEMPOOL0 buffer 2304 pool 32K cache 256 cpu 0
 2 |
 3 | link INVI port 1 rxq 1 128 MEMPOOL0 txq 1 512 promiscuous off
 4 | link OUTVI port 0 rxq 1 128 MEMPOOL0 txq 1 512 promiscuous off
 5 |
 6 | pipeline PIPELINE0 create 0
 7 |
 8 | pipeline PIPELINE0 port in 0 link INVI rxq 0 bsz 1
 9 | pipeline PIPELINE0 port out 0 link OUTVI txq 0 bsz 1
10|
12| pipeline PIPELINE0 build int.spec
13| pipeline PIPELINE0 table tb_int_inst_0003 update 003-table none none
14|
15| thread 1 pipeline PIPELINE0 enable
```

Výpis 5.1: Příklad konfiguračního souboru

použijí vždy a nemusí se provádět žádná další akce. Pro ostatní je pak potřeba jejich připojení pomocí pomocného skriptu. Případně lze aplikaci nastavit, které karty má použít. Můžeme také použít virtuální rozhraní vytvořená přímo při spuštění DPDK aplikace. Tato rozhraní mají potom vyšší čísla. Dalším zajímavým nastavením linky může být vypnutí nebo zapnutí promiskuitního módu.

2. Druhým důležitým nastavením, kterému je potřeba věnovat pozornost, je *pipeline*. Tu je nejprve potřeba vytvořit příkazem CREATE, způsobem ukázaným na řádku šest. Poté jí můžeme přiřadit vstupní a výstupní linky, řádky osm a devět. Jedna linka může být zároveň vstupní i výstupní, jen je zapotřebí tyto porty číslovat od nuly, jinak interpret zahlásí chybu se spec souborem[3]. I zde je potřeba zvýšit pozornost při konfiguraci a to zejména při nastavování parametru *bsz* (Burst sizes), kdy na některých kartách musí mít určitou velikost, většinou to musí být hodnota dělitelná čtyřmi či osmi. Pokud tuto hodnotu nenastavíme podle požadavků, nebudou nám programem procházet pakety.

3. Posledním krokem konfigurace je specifikace a nahrání konfiguračního souboru, vygenerovaného P4C-DPDK překladačem, řádek dvanáct. Případně můžeme vložit i soubory pro nastavení tabulek, jak je ukázáno na řádku třináct. Musíme jednak definovat, kterou tabulku chceme aktualizovat (*tb_int_inst_0003*) a také, v jakém souboru se nachází konfigurace (*003-table*).

Omezení implementace

1. Jelikož aplikace lze ovládat z kontrolní vrstvi jen omezeně, není k dipozici dokumentace ani žádné příklady, je program implementován jako šablona P4 kódu, do kterého se při generování skriptem v jazyce python přidají další informace. Při generování finálního kódu se použije pro zapnutí zvoleného uzlu nastavení výchozí akce příslušných tabulek. Pokud tedy chceme změnit funkci uzlu, musíme znovu přeložit a spustit aplikaci, to sice není ideální, ale na druhou stranu samotný překlad netrvá dlouhou dobu.

[3] Soubor vygenerovaný překladačem

Druhým řešením by mohlo být nastavování tabulek v CLI souboru, ale ani toto neodstraní problém, kdy musíme opětovně spouštět aplikaci, navíc nastavení opět nejsou zdokumentována a mohlo by to způsobit další problémy. Navíc první řešení nabízí lepší možnost testování kódu bez nutnosti kontrolovat, zda máme paket se správnými vstupními parametry.

2. Nelze použít *egress pipeline*, neboť ji překladač nesprávně generuje. Stejná chyba nastane pokud chceme rozdělit program na více částí. Ideálně by se nám hodilo mít každý kontrolní blok ve speciálním souboru, čímž by se výrazně zlepšila přehlednost programu. Navíc kvůli tomu není možné získávat metadata, která lze opatřit pouze při použití *egress* části, například výstupní časová značka, porty a latenci zařízení. Také se veškeré akce, které se v programu pro BMv2 prováděly v *egress* části, musely pro tuto implementaci přesunout do *ingress*. Toto se týká především funkcionality přidávání metadat, která v původní verzi nejprve v *ingress* nastaví své parametry a poté v *egress*, až zná všechna metadata, je začne podle masky doplňovat. V tabulce 5.1 můžeme vidět, která data lze pomocí této implementace podporovat.

3. V základní verzi překladače a interpretu sice lze použít *ingress* časovou značku, to znamená že příslušné políčko vnitřních metadat existuje, ale ve výchozím stavu vrací pouze nuly. Proto byla jak do překladače, tak do interpretu implementována nová instrukce, díky které lze do tohoto pole metadat, nahrát alespoň čas systémových hodin.

4. Aby bylo možné získávat ID portů, musí být použit speciální typ PortId_t, který je třiceti dvou bitový, na rozdíl od specifikace, kde je pouze šestnácti bitový. Tím pádem vzniká problém na sink uzlu. Pokud poslední reportovací uzel nebude také z této implementace, může nesprávně získat naměřená data nejen tohoto uzlu, ale i všech po něm následujících. Jedním řešením by mohl být příznak, podle kterého by ostatní aplikace poznaly, že se v paketu vyskytuje speciální hlavička.

5. Dalším omezením překladače je, že vyžaduje, aby všechna pole v hlavičkách měla velikost dělitelnou osmi. Z tohoto důvodu se tedy některá menší políčka musela spojit do jednoho většího celku. Minimální velikost polí nám občas ztěžuje práci, neboť kvůli tomu nelze k některým hodnotám přistupovat přímo a je zapotřebí použít logické operace a maskování.

6. Možná ještě větší problém způsobuje nemožnost přetypování. Překladač tuto možnost sice nabízí, podaří se nám tedy takový kód přeložit. Problém ovšem nastává až při interpretaci. Překladač k přetypování používá instrukci *cast*, kterou ale interpret nezná. Interpret poté nahlásí chybu na posledním řádku souboru se specifikací a programátor pak musí zdlouhavě hledat co se vlastně stalo. Detektivní práci výrazně ztěžuje překladač, který provádí různé optimalizace našeho kódu. Může se pak lehce stát, že pro dva programy, kde v obou použijeme přetypování, bude jeden fungovat a druhý se nespustí. Toto bude s velkou pravděpodobností způsobeno tím, že v jednom případě se přetypování použít nemuselo a překladač instrukci vyhodil, aby optimalizoval kód, ale v druhém případě nám zůstane a bude zdrojem problémů.

7. Další práci přidělala podpora kontrolních součtů. Do konstrukce pro počítání se musejí hlavičky vkládat po jednotlivých polích, zároveň se jejich hodnoty začnou postupně sčítat a to i v případě, že za sebou následují dvě osmibitové hodnoty.

Feature	State
Switch ID	OK
ID Portu	OK
ID Portu úrovně 2	Nelze
Vstupní časová značka	Použit systémový čas
Výstupní časová značka	Nelze, *egress pipeline* nelze použít.
Latence zařízení	Nelze, protože neznáme obě časové značky.
Zaplněnost front	PSA architektura zatím nedefinuje mechanismus pro získání těchto dat [3].
Využití výstupní TX linky	PSA architektura zatím nedefinuje mechanismus pro získání těchto dat [3].

Tabulka 5.1: Metadata

8. Protože nelze jednoduše použít pomocnou proměnnou, která by nám přidávala zarovnání, musely být některé hodnoty nahrány do pomocných proměnných. Některé byly ještě více pospojovány než bylo potřeba kvůli omezení na minimální velikost pole zmíněném výše. V současné době by měly fungovat oba kontrolní součty, IP i UDP.

9. Jedno z větších omezení se týká i *sink* uzlu. Vztahuje se k tomu, že nefunguje klonování, tím pádem z posledního uzlu můžeme poslat jen jeden paket a musíme zvolit mezi rámcem s naměřenými informacemi nebo originálním paketem, který by se po odstranění metadat, poslal k původnímu cíli. Koncový uzel má ještě jedno omezení, nedokáže správně odstranit všechna vložená metadat, vyjme pouze informace vložené samotným *sink* uzlem. K plné funkci by bylo potřeba získat hlavičky všech tranzitních uzlů, mohli bychom toho docílit použitím zásobníku hlaviček, ale jedná se opět o konstrukci, kterou překladač a interpret nejsou schopny zpracovat.

10. Protože nelze v klíči tabulky použít více různých hlaviček, museli být některé tabulky, ve kterých jich bylo využito více, upraveny. Toto omezení ovlivňuje především rozlišování toků pro aktivaci zdrojového uzlu. Ve výsledku ale implementovanou verzi toto omezení extrémně neovlivňuje, neboť výběry akcí tabulek jsou prováděny pomocí výchozích akcí i z tohoto důvodu.

11. Posledním a méně závažným rozdílem od implementace pro BMv2 je, že nelze zadávat konstantní záznamy tabulek přímo v P4. Toto, ale nezpůsobuje vážné problémy, neboť stejného efektu lze dosáhnout pomocí nastavení přímo při konfiguraci interpretu. V praxi to znamená, že přidáme pár řádků do CLI souboru a dosáhneme stejného výsledku.

Konfigurace přes kontrolní vrstvu

Kontrolní vrstvu lze částečně ovládat CLI souborem. Můžeme do ní zasahovat i za běhu a to pokud se připojíme pomocí *telnet 0.0.0.0 8086*.

CLI soubor slouží z větší části ke konfiguraci počátečního nastavení aplikace. Chceme v něm spíše než samotné P4 tabulky nastavovat linky, atd. viz dříve v této kapitole. Tabulky lze přes něj ovládat pouze tak, že mu řekneme, kde má hledat jejich konfigurační soubor. Konfigurační soubor aplikace vždy vyžaduje. Pokud ho nepředáme parametrem, přepne se do interaktivního režimu a je nutno parametry zadávat do konzole ručně.

Další akce přímo přes kontrolní vrstvu nejsou většinou možné. Funkčnost INT aplikace to z větší části neovlivňuje, jedná se spíše o záležitost pro usnadnění používání.

5.4 Implementace pro P4C-XDP

Pro tuto platformu nelze INT implementovat pouze za použití překladače, protože při kompilaci do bajt-kódu začne *clang* upozorňovat, že program překračuje povolenou velikost. Toto se začne dít jakmile začneme přidávat *emit* příkazy. Nenapadlo mě efektivní řešení, jak toto omezení obejít, proto výsledný program, ačkoli se ho zdařilo napsat způsobem, který se podaří přeložit, nelze spustit. To znemožňuje program otestovat a tím pádem nemůže být tato implementace prohlášena za funkční řešení.

Můžeme si, ale alespoň shrnout některé věci, které se mi i přesto podařilo vyzkoušet. Jedním z pozitiv je, že dokážeme získávat alespoň časové značky a to pomocí funkce *bpf_ktime_get_ns*, jež vrací čas od spuštění systému v nanosekundách. Při psaní aplikace pro tuto platformu je také nutné dát si pozor na několik věcí:

1. První z nich je, že pokud jsou v tabulkách použity klíče, musíme přidat k její definici, kromě standardních položek jako seznam akcí, klíč atd. i další položku a to jakým způsobem má být tato tabulka implementována. V přiložených příkladech je nejčastěji zmiňována a používána hodnota *hash_table(64)*. U definice tabulek je ještě potřeba zmínit, že pokud nenastavíme žádnou výchozí akci, nastaví se automaticky na NoAction, což je samozřejmě správně, ale poté nastane problém při překladu do bajt-kódu, kdy překladač nahlásí několikanásobnou definici právě akce NoAction.

2. Druhou věcí, jíž potřebujeme věnovat pozornost je, že se v programu musíme kdekoli v *ingress Match-Action* části definovat dvě položky a to *xout.output_port* a *xout.output_action*, pokud tak neučiníme, program nám nebude fungovat. Funkce *xout.output_port* definuje výstupní port a *xout.output_action*, co se má s paketem stát na konci zpracování.

Překladač má také řadu drobnějších chyb, ale některé se špatně hledají a jiné by možná nešly při dalším pokusu o napsání INT aplikace lehce obejít. Jedná se například i o syntaktické chyby při generování kódu v jazyce C, kdy například pokud bychom chtěli ručně aktivovat hlavičku (validovat) už v *parser* bloku funkcí *setValid*, překladač nepřidá na konec řádku středník ani odřádkování a vznikne tak nevalidní kód, který *clang* nedokáže přeložit.

XDP potřebuje pro spuštění některé specifické parametry (kernel flags). Tím pádem bylo potřebné program testovat na Ubuntu verzi 20.10.

5.5 Implementace pro P4C-eBPF

Program napsaný pro překladač eBPF dovoluje pouze filtrovat pakety a jak je naznačeno v tabulce 4.1, nedovoluje provádět žádné modifikace rámců. To znemožňuje napsat komplexnější aplikaci, kterou INT je. Tudíž jsem se ani nepokoušela napsat P4 program pro tento překladač.

5.6 Implementace pro P4C-uBPF

Jelikož uBPF překladač už je více podobní XDP překladači a podporuje i úpravy paketů, chtěla jsem vyzkoušet zprovoznit alespoň částečnou implementaci. Při pokusech otestovat

program jsem, ale narazila na problém, kdy aplikaci lze spustit pouze pomocí P4rt-OVS, který je součástí řešení *Open vSwitch* a není tím pádem univerzální.

5.7 Implementace pro T4P4S

Implementaci pro tento překladač se věnoval převážně kolega Ing. Mário Kuka. Aplikace v současné době podporuje pouze funkci zdrojového a tranzitního uzlu. Implementace koncového uzlu ještě není zcela otestována. Program je psán v $P4_{16}$ pro V1 model. K výběru, který typ uzlu bude použit musí dojít před překladem P4 kódu, změnou konfiguračního souboru. Návod na instalaci a spuštění je v GEÁNT repositáři[4].

Co se týká omezení implementace:

1. I zde funguje egress pipeline a program se tak v tomto ohledu neliší od referenční implementace. Problémy byly ale u časových značek, které ve výchozí verzi nejdou použít a musely být také do-implementovány. Nyní se přidává hodnota systémového času.

2. Druhý problém se projeví při používání aritmetických instrukcí, kdy program pracuje nesprávně, pokud se použijí položky větší než třicet dva bitů

Kontrolní vrstvu lze ovládat, ale pouze pomocí kódu psaného v jazyce C. Opět nejsou podporovány všechny konstrukce a její použití je nepohodlné. Navíc lze program přes kontrolní vrstvu ovládat jen v jedné fázi spouštění P4 aplikace. Kostra kontrolního programu je zobrazena v ukázce 5.2. Jediné čemu je potřeba věnovat pozornost je funkce *init()*, kde si uživatel může naprogramovat vlastní činnost kontroléru.

5.8 Shrnutí

Ze všech platforem se INT podařilo implementovat pouze pro BMv2, P4C-DPDK a T4P4S. V tabulce 5.2 můžeme vidět srovnání těchto platforem z pohledu funkcionality definované na začátku této kapitoly. Můžeme vidět, že z hlediska splnění požadavků je na tom platforma BMv2 nejlépe, podporuje většinu metadat. Navíc podporuje klonování paketů. Jak bylo řečeno dříve, nejedná se ale o řešení určené do produkce.

[4] https://github.com/GEANT-DataPlaneProgramming/int-platform-dpdk/tree/main/int_t4p4s_dpdk

```
// SPDX-License-Identifier: Apache-2.0
// Copyright 2016 Eotvos Lorand University,
Budapest, Hungary
controller c;
extern void notify_controller_initialized();

void dhf(void* b) {
}
void do_some_actions() {
    // user defined actions
}
void init() {

    do_some_actions();
    notify_controller_initialized();
}

int main(int argc, char* argv[])
{
    printf("Create and configure controller...\n");
    c = create_controller_with_init(11111, 3, dhf, init);

    printf("Launching controller’s main loop...\n");
    execute_controller(c);

    printf("Destroy controller\n");
    destroy_controller(c);

    return 0;
}
```

Výpis 5.2: Kostra kontrolního programu T4P4S

Vlastnost	BMv2	P4C-DPDK	T4P4S
Klonování	ANO	NE	NE
Switch ID	ANO	ANO	ANO
ID Portu	ANO	ANO	ANO
ID Portu úrovně 2	NE	NE	NE
Vstupní časová značka	ANO	ANO*	ANO*
Výstupní časová značka	ANO	NE	ANO*
Latence zařízení	ANO	NE	ANO
Zaplněnost front	NE	NE	NE
Využití výstupní TX linky	NE	NE	NE
Egress	ANO	NE	ANO
Časové značky	Čas od spuštění	Softwarový čas	Softwarový čas
Kontrolní součty	ANO	ANO	ANO

*Není součástí ve výchozím stavu, ale muselo být do-implementováno

Tabulka 5.2: Srovnání INT implementací [6]

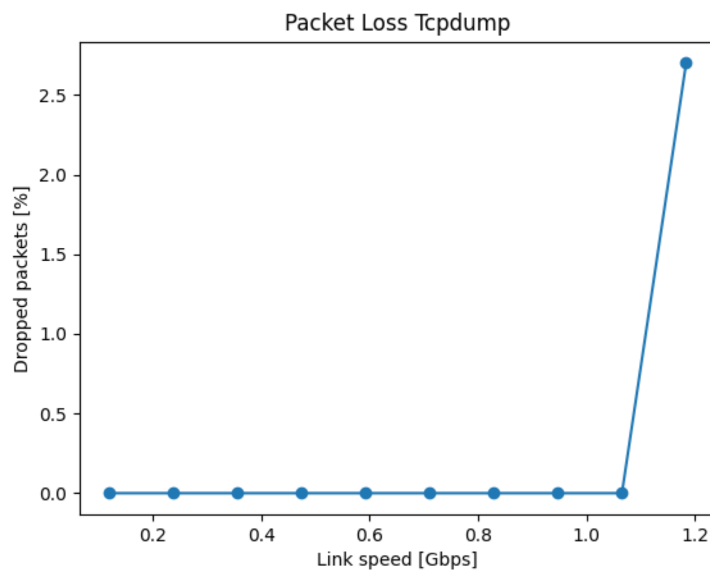
Kapitola 6

Testovaní INT aplikace

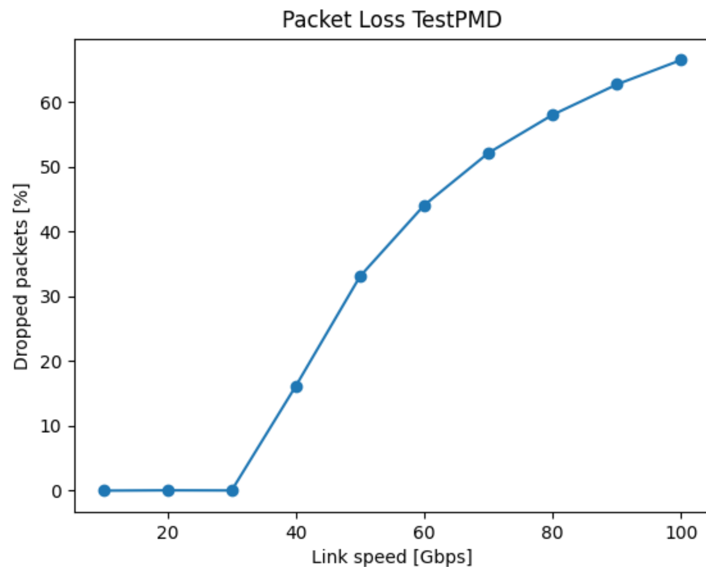
Byly provedeny celkem dva testy. Jeden pro měření rychlosti jednotlivých implementací a druhý pro měření přesnosti přiřazování časových značek. K měření bylo použito několik pomocných aplikací:

1. Tcpdump, je aplikace pro sledování a zachytávání provozu. Bylo potřebné změřit, kolik paketů zvládne zpracovat. Výsledky tohoto měření jsou zobrazeny v grafu 6.1. Je vidět, že začne zahazovat pakety až při rychlosti kolem 1 Gb/s, kdy nezpracuje pouze asi tři procenta paketů.

2. DPDK-TestPMD, je aplikace pro testování přeposílání paketů pomocí DPDK. Bude tedy sloužit jako reference pro měření rychlosti. Graf 6.2 zobrazuje propustnost této aplikace. Jak je vidět, aplikace zvládne zpracovávat pakety bez zahazování až do rychlosti 30 Gb/s.



Obrázek 6.1: Propustnost tcpdump



Obrázek 6.2: Propustnost DPDK-TestPMD

6.1 Použitý hardware

Pro měření byl použit stroj s názvem OVS. Současně byl použit i méně výkonný stroj Veltliner, protože všechna měření rychlosti nebylo možné na stroji OVS z technických a časových důvodů provézt. Parametry obou zařízení jsou na ukázce 6.1. Byly získané pomocí příkazů *hostnamectl*, *lscpu* a *cat /proc/meminfo*.

Pro generování provozu byl použit stroj Spirent[1] při jednom měření i aplikace *tcpreplay* přeložená ze zdrojového kódu společně s *netmap*, aby dokázala posílat pakety většími rychlostmi.

Měření probíhala na síťové kartě Mellanox. Měření na stroji Veltliner pak s kartou Intel. Specifikace obou karet můžete vidět na ukázce 6.2.

6.2 Použité testovací schéma

Většina měření byla prováděna na OVS stroji. Byla provedena celkem dvě měření. Jedno se zaměřuje na maximální rychlost, kterou je implementace schopna pracovat, druhé pak na měření přesnosti přidělování časových značek.

Měření rychlosti

Toto měření probíhalo nejprve na stroji Veltliner s Intel kartou. Naznačení testovacího procesu můžete vidět na obrázku 6.3.

Nejprve bylo potřeba vytvořit sady vstupních paketů. Jednotlivé sady se odlišovaly délkou paketů, v jedné sadě tedy pakety měly vždy stejnou velikost. Uvnitř sady měl ideálně každý rámec jiné IP adresy a UDP čísla portů, aby DPDK aplikace mohly pakety rozdělovat do různých front. Při samotném měření se vzala vždy jednu sadu paketů a poslali ji pomocí

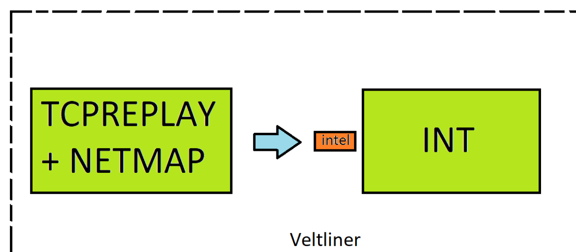
[1]Detailní specifikace stroje: https://assets.ctfassets.net/wcxs9ap8i19s/7mjZqW5guntME8DD0gf27g/dcc39be517f3296d4ada20b3ecaec900/SPT-N11U_Mainframe_Chassis_Datasheet.pdf

```
OVS
========================================
System:
    OS: Oracle Linux Server 8.5
    Kernel: Linux 4.18.0-240.el8.x86_64
    Architecture: x86-64
CPU:
    Model: Intel(R) Xeon(R) Gold 6230N CPU @ 2.30GHz
    Topology:
    Speed: 1000 MHz
CPU(s): 80
RAM: 376.28 GiB

Veltliner
========================================
System:
    OS: Ubuntu 20.04.3 LTS
    Kernel: Linux 5.4.0-107-generic
    Architecture: x86-64
CPU:
    Model: Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz
    Speed: 1200 MHz
CPU(s): 24
RAM: 31.35 GiB
```

Výpis 6.1: Parametry OVS a Veltliner stroje

```
Mellanox
========================================
Typ: Mellanox Technologies MT28800 Family [ConnectX-5 Ex]
Driver: mlx5
Rychlost: 100 Gb/s

Intel
========================================
Typ: Intel Corporation Ethernet Controller X710 for 10GbE SFP+ (rev 02)
Driver: i40e-2.16.11
Rychlost: 10 Gb/s
```

Výpis 6.2: Použité síťové karty



Obrázek 6.3: Testovací schéma pro měření rychlosti na Veltlineru za použití aplikace tcpreplay

aplikace *tcpreplay* na vstupní port INT programu. Ze statistik *tcpreplay* jsme potom odvodili jakou rychlost aplikace zvládne. Tabulka 6.1 ukazuje některé naměřené hodnoty.

	Velikost paketů [B]				
Aplikace	64	128	256	512	1024
DPDK-TestPMD	6.4	8.4	9.1	9.5	9.7
P4C-DPDK	0.5	1.3	2.4	4.1	5.5
T4P4S	0.1	0.2	0.5	0.9	1.8
T4P4S s 9 frontami	1.1	2.1	4.3	8.1	9.7

Tabulka 6.1: Rychlosti DPDK aplikací při použití tcpreplay v [Gbps]

Můžeme vidět, že na kartě Intel, která by měla zvládnout rychlost až 10 Gb/s, se k této rychlosti blíží pouze testovací aplikace *DPDK-TestPMD*, která je zde uvedena jako reference, a INT program vygenerovaný překladačem T4P4S[2], ale pouze v případě, že je spuštěn na více vláknech a frontách. Aplikace získaná kompilátorem P4C-DPDK nedovoluje nastavení front, proto nemohlo být otestováno.

Druhé doplňkové měření probíhalo již na stroji OVS s Mellanox síťovou kartou za použití stroje Spirent, který posílal pakety na vstupní rozhraní aplikace, ale následně i měřil rychlost na výstupu. Schéma tohoto testování je zobrazeno na obrázku 6.4.

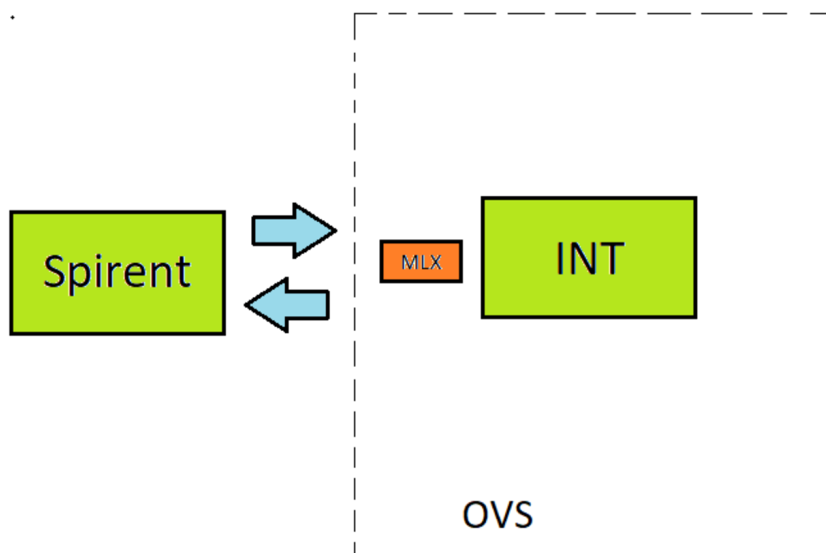
Měření probíhalo obdobným způsobem jako na stroji Veltliner. Strojem Spirent byly poslány pakety na vstupní rozhraní DPDK aplikace, ta zároveň používala stejné rozhraní pro výstup, tudíž se provoz dostal zpět na Spirent, kde byly odečteny naměřené hodnoty. Výsledky jsou vidět v tabulce 6.2.

	Velikost paketů [B]	
Aplikace	128	1024
DPDK-TestPMD	24.7	47.2
P4C-DPDK	1.8	10.4

Tabulka 6.2: Rychlosti DPDK aplikací při použití stroje Spirent v [Gbps]

Můžeme vidět, že se rychlost aplikace generované překladačem P4C-DPDK trochu zvýšila, ale stále nedosahuje převratných hodnot.

[2] 9 front bylo použito z důvodu limitu Veltliner stroje



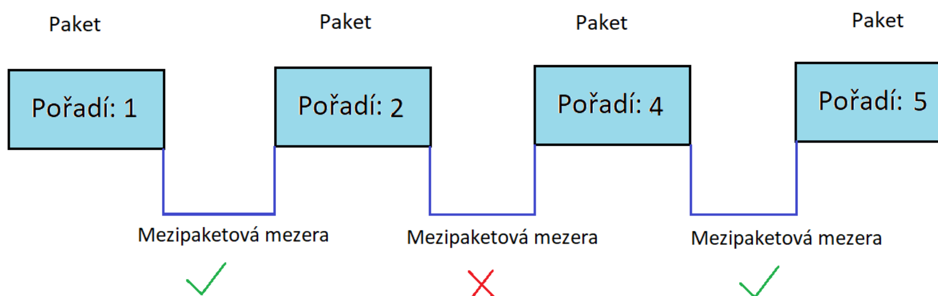
Obrázek 6.4: Testovací schéma pro měření rychlosti na OVS za použití stroje Spirent

Měření přesnosti časových značek

Účelem měření bylo zjistit přesnost přidělování časových značek. Přes měřenou aplikaci byl poslán známí počet paketů o známé velikosti s námi zvolenou mezi-paketovou mezerou. Následně se tato teoretická mezera porovnala s naměřenou a provedlo se vyhodnocení. Výsledky jsou zobrazeny jako rozdíl naměřené a teoretické hodnoty.

V každém měření bylo posláno milion paketů o velikosti 128 bajtů. Jednotlivá měření se od sebe odlišovala rychlostí, kterou byly tyto pakety odesílány. Rychlost byla zadávána v rámcích za sekundu (r/s).

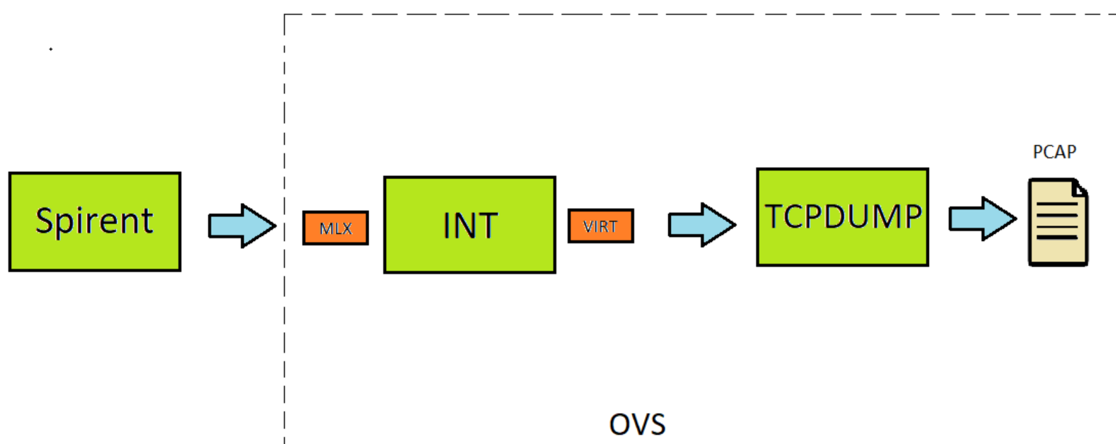
Důležité pro tento typ měření je i to, aby pakety následovaly za sebou. Pokud se stane, že se některý paket ztratí, nesmí se do výpočtu započítat mezera paketů, mezi kterými dojede ke ztrátě. Situace je zobrazena na obrázku 6.5. Tento požadavek do měření přináší další kritérium. Tím je procento validních mezer. Jelikož jsou používány UDP pakety, které nemají sekvenční čísla, byly místo nich použity čísla portů, jež se přidělovaly při generování.



Obrázek 6.5: Validita mezi-paketových mezer

Pro každou implementaci bylo provedeno deset měření. Začalo se s rychlostí 100 000 r/s, což je přibližně 0.1 Gb/s a pokračovalo se s krokem 100 000 r/s až na hodnotu 1 000 000 r/s, což je asi 1.2 Gb/s.

Na obrázku 6.6 je zobrazeno schéma, které bylo použito pro měření přesnosti časových značek. Jak je zde naznačeno, generování paketů probíhá přes zařízení *Spirent*. Rámce se pošlou přes Mellanox kartu, kde čeká příslušná měřená INT aplikace, která pakety zpracovává a posílá na výstupní virtuální rozhraní. Zde je provoz zachycen aplikací *tcpdump*, jež všechny přijaté pakety uloží do *pcap* souboru.



Obrázek 6.6: Testovací schéma pro měření přesnosti časových značek

Takto byly zachyceny pakety pro všechny zvolené rychlosti. Po získání potřebných hodnot se můžeme přesunout na vyhodnocení výsledků. K tomu nám posloužily dva pomocné programy:

1. Jeden pro získání rozdílu mezi časovými značkami a celkového počtu zachycených paketů, *sniffer.c*. Program čte pakety z vybraného *pcap* souboru, získá časovou značku a sekvenční číslo. Pokud dva pakety za sebou následují správně, odečtou se časové značky obou paketů a zaznamená se tento rozdíl. Pokud nenásledují, rozdíl se nevypočítá, ale zvýšíme počítadlo nevalidních mezer a pokračujeme na další paket.

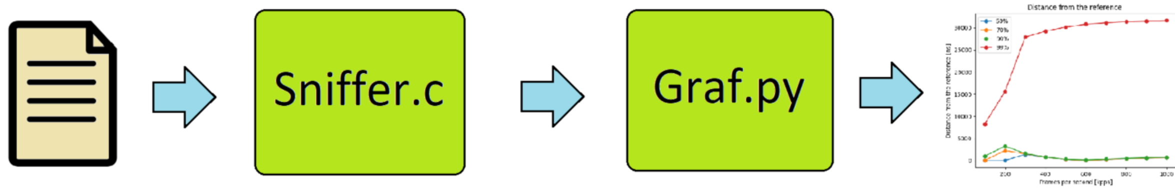
2. A druhý, *graf.py*, pro výsledné zpracování výsledků do podoby grafů. Program bere na vstupu data, která byla vygenerována předchozím programem.

Obrázek 6.7 ukazuje názorně postup získávání výsledků.

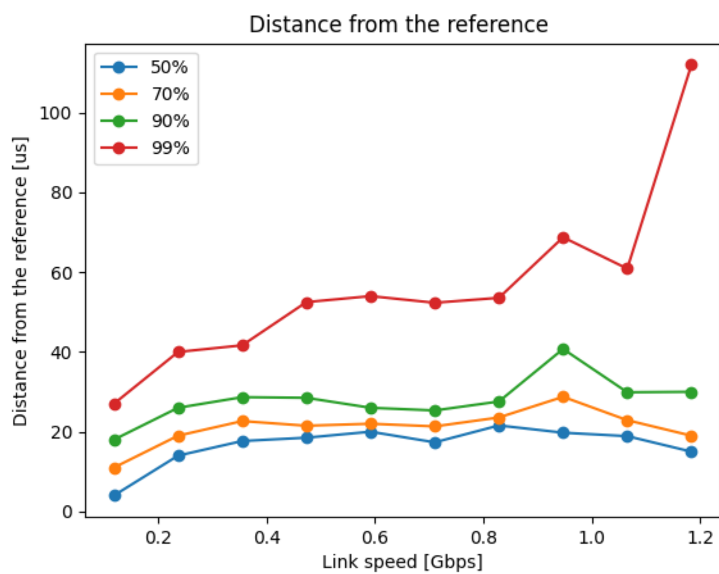
Ještě je potřeba dodat, že aby bylo možné výsledky lépe porovnávat, byla všechna měření prováděna jen při použití jedné DPDK fronty na jednom vlákně.

Následující grafy (6.8, 6.9, 6.10) zobrazují výsledky měření rozdílu časových značek pro jednotlivé implementace INT.

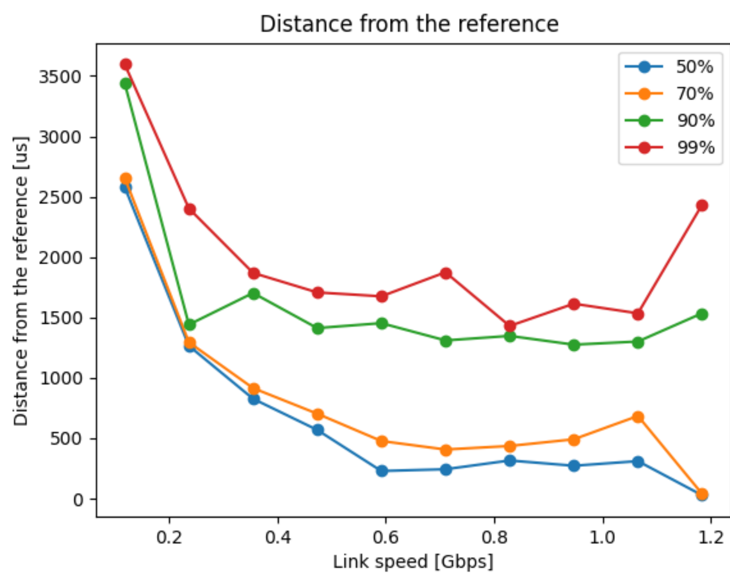
Výrazně nejhorší, co se týče přesnosti časových značek, je implementace pro P4C-DPDK. Je potřeba podotknout, že má na druhou stranu nejlepší propustnost. Tu můžeme sledovat na grafech 6.11. To samé se týká i parametru procenta validních paketů. Graf je na obrázku 6.12. Vysoké procento zahozených paketů u DPDK implementací může být způsobeno i tím, že jako výstupní rozhraní z aplikací je použito virtuální rozhraní. Druhým důvodem může být i to, že aplikace nejsou spuštěny přímo na OVS stroji, ale jsou zprovozněna ve virtuálním prostředí, kde může virtualizace přidávat další režii.



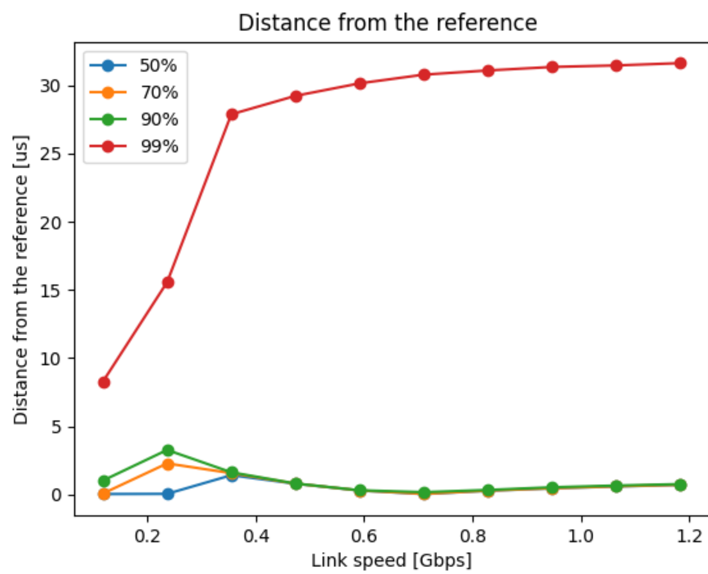
Obrázek 6.7: Schéma zpracování naměřených výsledků pro přesnost časových značek



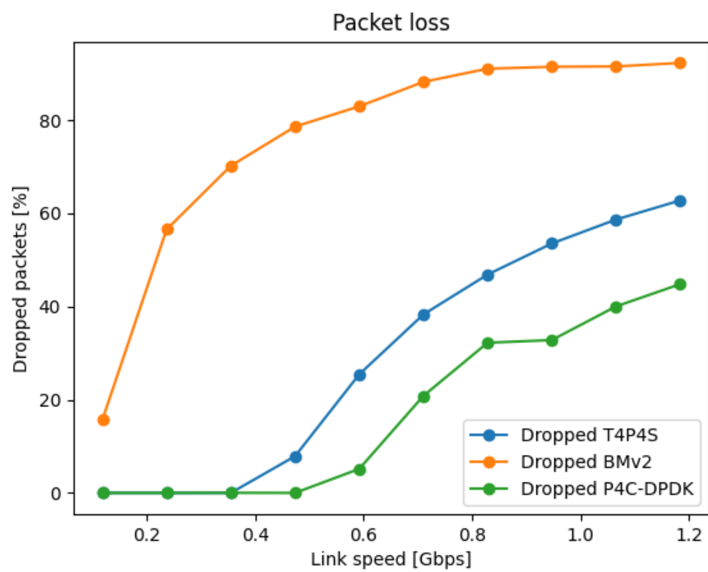
Obrázek 6.8: Rozdíl časových značek BMv2



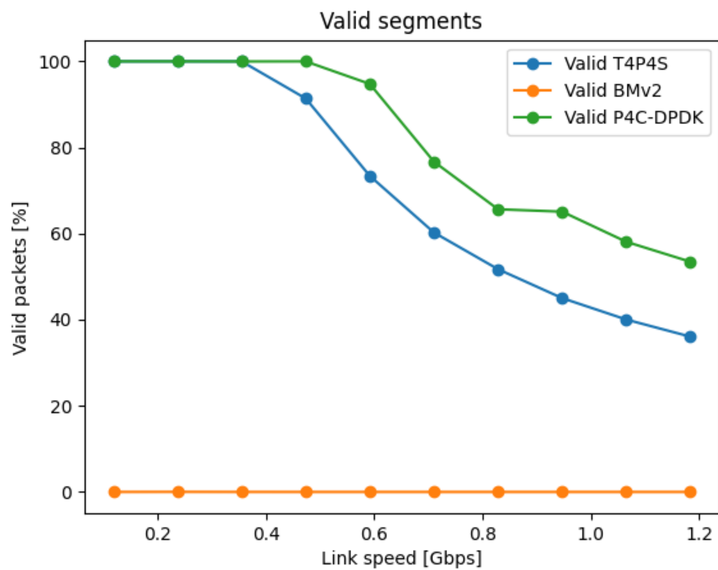
Obrázek 6.9: Rozdíl časových značek P4C-DPDK



Obrázek 6.10: Rozdíl časových značek T4P4S



Obrázek 6.11: Propustnost implementací



Obrázek 6.12: Validní segmenty pro měřené platformy

Kapitola 7

Závěr

Cílem práce bylo seznámit se s dostupnými překladači pro P4 a následně se pokusit implementovat In-band Network Telemetry. Byla popsána funkcionalita INT, P4 a překladačů tohoto jazyka pro různé platformy. V dalších kapitolách jsou pak diskutovány jednotlivé implementace, jejich omezení případně, proč nelze dané řešení použít.

Byly shrnuty přednosti a omezení již funkčních implementací. Konkrétně T4P4S pro platformu DPDK a P4C-BMv2, který lze spustit buď v emulátoru Mininet, nebo pomocí behaviorálního simulačního modelu *Simple Switch*. Podařilo se implementovat i aplikaci přeložitelnou kompilátorem P4C-DPDK, jež lze spustit pomocí softwarového DPDK přepínače SWX. Program pro XDP, uBPF a eBPF se nepodařilo zprovoznit. V dalším kroku bylo provedeno měření propustnosti a přesnosti přiřazování časových značek jednotlivými programy. Z testovaných softwarových implementací nejlepší výkon podala aplikace přeložená kompilátorem T4P4S.

Jelikož se většina popsaných překladačů dále vyvíjí, může se v budoucnu stát, že bude možné INT aplikaci napsat pro větší množství platforem. V této práci jsme se věnovali převážně open source platformám, jako další pokračování by tedy mohlo být zajímavé, vyzkoušet komerční platformy, například Tofino.

Literatura

[1] Andy Fingerhut, A. B. *BEHAVIORAL MODEL (bmv2)* [online]. Únor 2022 [cit. 2022-02-26]. Dostupné z: https://github.com/p4lang/behavioral-model#running-your-p4-program.

[2] Andy Fingerhut, A. B. *The BMv2 Simple Switch target* [online]. Leden 2022 [cit. 2022-02-26]. Dostupné z: https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md.

[3] Andy Fingerhut, R. S. *PSA.mdk* [online]. Duben 2021 [cit. 2022-04-26]. Dostupné z: https://github.com/p4lang/p4-spec/blob/main/p4-16/psa/PSA.mdk.

[4] Bas, A. *Performance of bmv2* [online]. Listopad 2019 [cit. 2022-04-09]. Dostupné z: https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md.

[5] Calin Cascaval, D. D. *P4 Architectures* [online]. Prosinec 2020 [cit. 2022-02-26]. Dostupné z: https://opennetworking.org/wp-content/uploads/2020/12/p4-ws-2017-p4-architectures.pdf.

[6] Chris Dodd, A. B. *V1modle* [online]. Prosinec 2021 [cit. 2022-04-26]. Dostupné z: https://github.com/p4lang/p4c/blob/main/p4include/v1model.p4.

[7] Consortium, T. P. L. *P416 Language Specification* [online]. 1.2.2. The P4 Language Consortium, květen 2021 [cit. 2022-02-26]. Dostupné z: https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf.

[8] Dodd, C. *Documenting v1model more fully* [online]. Duben 2019 [cit. 2022-02-26]. Dostupné z: https://github.com/p4lang/p4-spec/issues/756#issuecomment-486754378.

[9] Fungible. *Tutorial: Separating Data Plane and Control Plane: Why it Matters?* [online]. Březen 2020. Dostupné z: https://www.youtube.com/watch?v=MCssZUGQeAo&ab_channel=Fungible.

[10] Group, T. P. A. W. *In-band Network Telemetry (INT) Dataplane Specification* [online]. 1.0. The P4.org Applications Working Group, duben 2018 [cit. 2022-04-09]. Dostupné z: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v1_0.pdf.

[11] Group, T. P. A. W. et al. *In-band Network Telemetry (INT) Dataplane Specification*. 2.1. Listopad 2020. Dostupné z: https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[12] Group, T. P. A. W. *P416 Portable Switch Architecture (PSA)* [online]. Working draft. The P4.org Architecture Working Group, duben 2021 [cit. 2022-02-26]. Dostupné z: https://p4.org/p4-spec/docs/PSA.html.

[13] Han Wang, R. S. *DPDK backend* [online]. Březen 2022 [cit. 2022-03-26]. Dostupné z: https://github.com/p4lang/p4c/tree/main/backends/dpdk.

[14] Ing. Filip Holík, P. a Ing. Sona Neradová, P. *Softwarově definované sítě* [online]. První. Univerzita Pardubice, červenec 2019 [cit. 2022-02-26]. ISBN 978-80-7560-235-0 (pdf). Dostupné z: https://dk.upce.cz/bitstream/handle/10195/73810/978-80-7560-235-0%20Sofawarove%20definovane%20site.pdf?sequence=1&isAllowed=y.

[15] kolektiv, A. F. a. *P4c* [online]. Únor 2022 [cit. 2022-02-26]. Dostupné z: https://github.com/p4lang/p4c.

[16] Laki, S. *T4P4S:When P4 meets DPDK* [online]. Září 2017 [cit. 2022-04-27]. Dostupné z: https://www.dpdk.org/wp-content/uploads/sites/35/2017/09/DPDK-Userspace2017-Day2-12-SANDOR_LAKI-T4P4S.pdf.

[17] Laki, S. a Vörös, P. *T4P4S: A multi-target P4 compiler framework* [online]. Červen 2019 [cit. 2022-02-26]. Dostupné z: http://p4.elte.hu/publications/p4-ws-2019.pdf.

[18] Mihai Budiu, T. O. *EBPF Backend* [online]. Prosinec 2020 [cit. 2022-02-26]. Dostupné z: https://github.com/p4lang/p4c/tree/main/backends/ebpf.

[19] Mihai Budiu, T. O. *Introduction to uBPF Backend* [online]. Květen 2020 [cit. 2022-02-26]. Dostupné z: https://github.com/p4lang/p4c/tree/main/backends/ubpf.

[20] Mihai Budiu, T. O. *P4c-uBPF-model* [online]. Květen 2020 [cit. 2022-02-26]. Dostupné z: https://github.com/p4lang/p4c/blob/main/backends/ubpf/p4include/ubpf_model.p4.

[21] Mihai Budiu, W. T. *Compiling P4 to XDP* [online]. Únor 2017 [cit. 2022-02-26]. Dostupné z: https://github.com/vmware/p4c-xdp/blob/master/doc/p4xdp-iovisor17.pdf.

[22] Mihai Budiu, W. T. *P4c-xdp* [online]. Březen 2022 [cit. 2022-02-26]. Dostupné z: https://github.com/vmware/p4c-xdp.

[23] Mihai Budiu, W. T. F. R. *Linux Network Programming with P4* [online]. Leden 2019 [cit. 2022-02-26]. Dostupné z: https://github.com/vmware/p4c-xdp/blob/master/doc/lpc18.pdf.

[24] Mihai Budiu, W. T. F. R. *Linux Network Programming with P4* [online]. Leden 2019 [cit. 2022-02-26]. Dostupné z: https://github.com/vmware/p4c-xdp/blob/master/doc/p4c-xdp-lpc18-presentation.pdf.

[25] Osiński, T. *P4c-ubpf: a New Back-end for the P4 Compiler* [online]. ONF, červen 2020 [cit. 2022-04-19]. Dostupné z: https://opennetworking.org/news-and-events/blog/p4c-ubpf-a-new-back-end-for-the-p4-compiler/.

[26] O'Connor, B. a Cascone, C. *P4 and P4Runtime basics* [online]. Leden 2019 [cit. 2022-02-26]. Dostupné z: https://opennetworking.org/wp-content/uploads/2019/10/NG-SDN-Tutorial-Session-1.pdf.

[27] Parniewicz, D. *P4 INT implementation for bmv2 switches running within mininet environment* [online]. Březen 2021 [cit. 2022-02-26]. Dostupné z: https://github.com/GEANT-DataPlaneProgramming/int-platforms/blob/master/platforms/bmv2-mininet/README.md.

[28] Parol, P. *P4 Network Programming Language—what is it all about?* [online]. Duben 2020 [cit. 2022-02-26]. Dostupné z: https://codilime.com/blog/p4-network-programming-language-what-is-it-all-about/.

[29] Pavel Benáček, V. P. *Jazyk P4 jako budoucnost SDN* [online]. Leden 2016. Dostupné z: https://www.cesnet.cz/2016/01/jazyk-p4/.

[30] Radostin Stoyanov, N. F. *P4 Implementing Basic Forwarding tutorial* [online]. Únor 2022 [cit. 2022-02-26]. Dostupné z: https://github.com/p4lang/tutorials/blob/master/exercises/basic/solution/basic.p4.

[31] Tan, L. et al. In-band Network Telemetry: A Survey. *Computer Networks* [online]. 2021, sv. 186, s. 107763. DOI: https://doi.org/10.1016/j.comnet.2020.107763. ISSN 1389-1286. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1389128620313396.

[32] Tu, W. *P4xdp-workflow.png* [online]. Únor 2017 [cit. 2022-02-26]. Dostupné z: https://github.com/vmware/p4c-xdp/blob/master/doc/images/p4xdp-workflow.png.

[33] Vörös, P., Horpácsi, D., Kitlei, R., Leskó, D., Tejfel, M. et al. T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors. In:. červen 2018 [cit. 2022-03-18]. DOI: 10.1109/HPSR.2018.8850752. Dostupné z: https://www.researchgate.net/publication/326652427_T4P4S_A_Target-independent_Compiler_for_Protocol-independent_Packet_Processors.

Příloha A

INT Metadata

INT pojem	Vysvětlivka
INT Header	Hlavička v paketu s INT informacemi
INT Packet	Paket s INT hlavičkou
INT Node	Zařízení schopné provozovat INT
INT Instruction	Říká, která metadata mají být přidána INT uzlem
INT Source	Důveryhodné zařízení, které vloží INT hlavičku
INT Sink	Důveryhodné zařízení, které odstraní INT hlavičku a metadata vložení INT uzly
INT Transit Hop	Důveryhodné zařízení, které získává metadata podle INT instrukcí
INT Metadata	Informace, které INT uzly vkládají do paketu podle instrukcí

Příloha B

Spuštění INT pro BMv2

Instrukce k nainstalování a spuštění INT pro BMv2 v aplikaci Simple Switch

- Nainstalujte behaviorální model pomocí návodu
- Nainstalujte P4 překladač pomocí návodu
- Přeložte INT aplikaci

```
p4c-bm2-ss --p4v 16 "<in.p4>" -o "<out.json>" -DBMV2
```

  Kde <in.p4> je P4 program, který chceme přeložit a <out.json> je přeložený kód, který budeme dále nahrávat do Simple Switche. Příklad:

```
p4c-bm2-ss --p4v 16 "int.p4" -o "int.json" -DBMV2
```

  P4 program je dostupný v githubovém repozitáři Geántu

- Spusťte Simple Switch aplikaci

```
sudo simple_switch -i 0@<in-ifc> -i 1@<out-ifc> <out.json>
```

  Kde <in-ifc> a <out-ifc> jsou vstupní a výstupní rozhraní a <out.json> je P4 program přeložený v předchozím kroku. Můžeme využít spoustu dalších přepínačů, které jsou popsané v dokumentaci pro Simple Switch.

  Příklad:

```
sudo simple_switch --log-console -i 0@enp7s0 -i 1@enp9s0 int.json
```

- Konfigurace programu přes kontrolní vrstvu. Nejprve spustíme *runtime_CLI.py* aplikaci, jež se nachází ve složce *tools* v repozitáři behaviorálního modelu.

  Poté můžeme náš program nastavit požadovaným způsobem. Následující příklad je pro spuštění zdrojového uzlu.

```
table_set_default tb_activate_source activate_source
table_set_default tb_int_source configure_source 4 6 4 0xCC00
table_set_default tb_int_transit configure_transit 1 1500
table_set_default tb_forward send_to_port 1
```