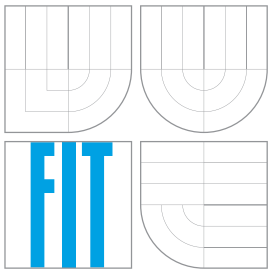
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

FILTRACE PAKETŮ VE 100 GB SÍTÍCH

PACKET FILTRATION IN 100 GB NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN KUČERA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KOŘENEK, Ph.D.

BRNO 2016

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Kučera Jan, Bc.**
Obor: Počítačové a vestavěné systémy
Téma: **Filtrace paketů ve 100 Gb sítích**
Packet Filtration in 100 Gb Networks
Kategorie: Počítačová architektura

Pokyny:

1. Nastudujte možnosti hardwarové akcelerační karty COMBO 100G.
2. Seznamte se s hardwarovými architekturami a algoritmy pro filtraci paketů.
3. Analyzujte dostupné algoritmy a hardwarové architektury pro filtraci paketů z pohledu časové a paměťové složitosti.
4. Navrhněte vhodný způsob filtrace paketů pro 100Gb sítě v FPGA. Při návrhu zohledněte možnosti paralelního zpracování v technologii FPGA a pokuste se dosáhnout vhodného kompromisu mezi paměťovou a časovou složitostí.
5. Ověřte vlastnosti navržené architektury na dostupných množinách filtračních pravidel.
6. Proveďte implementaci navrženého řešení s ohledem na mapování do technologie FPGA a analyzujte vlastnosti vytvořené implementace.
7. V závěru diskutujte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kořenek Jan, Ing., Ph.D.**, UPSY FIT VUT
Konzultant: Puš Viktor, Ing., Ph.D., CESNET
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Abstrakt

Diplomová práce se zabývá návrhem a implementací algoritmu pro filtraci paketů pro vysokorychlostní počítačové sítě. Hlavním cílem bylo vytvoření hardwarové architektury pro filtraci, která dosáhne vysoké kapacity ve smyslu počtu filtračních pravidel a umožní nasazení v sítích o rychlostech až 100 Gb/s. Návrh systému byl proveden s ohledem na možnost paralelního zpracování při implementaci v technologii FPGA a s cílem nalezení vhodného kompromisu mezi časovou a paměťovou složitostí algoritmu. Dosažené vlastnosti navržené architektury a vytvořené implementace byly následně ověřeny na dostupných množinách filtračních pravidel. Díky vysoce optimalizované architektuře a řetězenému zpracování bylo možné při implementaci dosáhnout vysoké pracovní frekvence (přes 220 MHz) a současně významně zredukovat paměťové nároky (v průměru o 72 % oproti porovnávaným algoritmům). Efektivní využití interní paměti dostupné přímo na čipu umožňuje s použitím FPGA uložení až pěti tisíc filtračních pravidel při zabrání pouze 8 % dostupné kapacity paměti. To vše při současném dosažení plné propustnosti linky 100 Gb/s.

Abstract

This master's thesis deals with the design and implementation of an algorithm for high-speed network packet filtering. The main goal was to provide hardware architecture, which would support large rule sets and could be used in 100 Gbps networks. The system has been designed with respect to the implementation on an FPGA card and time-space complexity trade-off. Properties of the system have been evaluated using various available rule sets. Due to the highly optimized and deep pipelined architecture it was possible to reach high working frequency (above 220 MHz) together with considerable memory reduction (on average about 72 % for compared algorithms). It is also possible to efficiently store up to five thousands of filtering rules on an FPGA with only 8 % of on-chip memory utilization. The architecture allows high-speed network packet filtering at wire-speed of 100 Gbps.

Klíčová slova

Filtrace paketů, klasifikace, vysokorychlostní sítě, 100 Gb/s, hardwarová akcelerace, FPGA.

Keywords

Packet Filtering, Classification, High-speed Networks, 100 Gbps, Hardware Acceleration, FPGA.

Citace

KUČERA, Jan. *Filtrace paketů ve 100 Gb sítích.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kořenek Jan.

Filtrace paketů ve 100 Gb sítích

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Kořenka, Ph.D., a že jsem uvedl všechny literární prameny a publikace, z nichž jsem čerpal.

.....
Jan Kučera
25. května 2016

Poděkování

Rád bych poděkoval především vedoucímu práce Ing. Janu Kořenkovi, Ph.D., za odborné vedení, ochotu a trpělivost při konzultacích. Velké poděkování patří také mému konzultantovi Ing. Viktoru Pušovi, Ph.D., za veškerou pomoc a cenné rady. Rád bych poděkoval také svým rodičům za jejich maximální podporu nejen při tvorbě této práce, ale i v průběhu mého celého magisterského studia.

Obsah

Kapitola 1

Úvod

Dnešní svět je charakteristický rychlým rozvojem ve všech oblastech lidské činnosti. Jinak tomu není ani v odvětví informačních technologií. Výpočetní technika a počítače v jakékoliv formě se staly zcela běžnou součástí našeho každodenního života a jsme na nich svým způsobem závislí. Největší rozmach nastal v období rozvoje počítačových sítí, obzvláště té největší – internetu. Počítačové sítě a internet byly od svého počátku navrženy a budovány pro propojení důvěryhodných počítačů. Charakter sítě se však od té doby podstatným způsobem změnil. Dnes je třeba nepřetržitě sledovat aktuální bezpečnostní hrozby a značné úsilí věnovat celkovému zabezpečení počítačových sítí.

Filtrace paketů je jedním ze základních mechanismů pro zajištění bezpečnosti počítačových sítí, který umožňuje zablokování nebezpečného síťového provozu. V řadě systémů se využívá v kombinaci s pokročilými technikami monitorování sítě a detekce podezřelé aktivity a anomálií, které umožňují identifikaci aktuálních hrozeb a útoků a zpětně zajišťují odpovídající úpravu pravidel pro filtraci provozu.

Filtrace paketů zahrnuje příjem paketu, analýzu a extrakci položek z hlaviček paketu, klasifikaci paketu a provedení akce přidružené k nalezenému pravidlu. Proces klasifikace a vyhledání odpovídajícího pravidla je přitom klíčovou a velmi výpočetně náročnou operací, která ovlivňuje celkovou rychlost a cenu výsledného systému.

S rostoucí rychlostí počítačových sítí rostou i požadavky na rychlost zpracování paketů. Síťová zařízení musí umožňovat zpracování neustálého proudu i těch nejkratších paketů a prováděná klasifikace musí být schopná dosáhnout plné propustnosti linky (tzv. wire speed), aby neomezovala šířku pásma. To je kritické především na páteřních linkách vysokorychlostních sítí s přenosovými rychlostmi 100 Gb/s a více. Druhým důležitým faktorem ovlivňujícím filtraci jsou neustále rostoucí požadavky na klasifikaci ve smyslu potřebného počtu klasifikačních pravidel, který souvisí s nárůstem počtu zařízení připojených k síti. Trendem je také provádění klasifikace podle daleko většího počtu položek z hlaviček paketů. Řešení celého problému filtrace a klasifikace paketů tak spočívá v nalezení kompromisu mezi podporovaným množstvím klasifikačních pravidel, jejich složitostí, dosahovanou propustností a celkovou cenou výsledného systému.

Dosažení dostatečné rychlosti a výkonu je u čistě softwarově orientovaného přístupu a implementace pro univerzální výpočetní procesory prakticky nemožné. Řešení problémů v oblastech vyžadujících vysoký výkon a rychlost zpracování se totiž neobejde bez využití hardwarové akcelerace. Takovou oblastí je i filtrace paketů ve 100 Gb sítích. Jednou z možností je použití technologie ASIC (Application-specific Integrated Circuits). Aplikačně specifické integrované obvody poskytují dostatečný výkon, jsou však velmi drahé, především z pohledu vysokých počátečních nákladů při návrhu masky. Vyplatí se tak pouze

při výrobě velkých sérií. U síťových zařízení se často využívají také asociativní paměti TCAM, které umožňují velmi rychlé vyhledávání, avšak nevýhodou je jejich vysoký příkon a cena vzhledem k dostupné kapacitě. Další možnou alternativou je technologie FPGA (Field Programmable Gate Arrays). Programovatelná hradlová pole sice ve srovnání s čipy ASIC nedosahují takové výkonnosti, tuto svou nevýhodu však plně vyváží možností rekonfigurace. Díky snadné rekonfigurovatelnosti nachází uplatnění také v oblasti výzkumu a akcelerace síťových aplikací. Nespornou výhodou FPGA je totiž jednodušší a rychlejší vývoj a s tím související rychlejší uvedení cílového produktu na trh.

Tato diplomová práce se zabývá návrhem algoritmu pro filtraci paketů na vysokorychlostních sítích s přenosovou rychlostí až 100 Gb/s. Cílem práce je navržení a implementace hardwarové architektury pro filtraci paketů na akcelerační síťové kartě s FPGA. Při návrhu je kladen důraz na dosažení maximálního počtu filtračních pravidel s cílem nalezení vhodného kompromisu mezi časovou a paměťovou složitostí algoritmu a s ohledem na možnost paralelního zpracování při implementaci v FPGA.

V rámci úvodních kapitol diplomové práce je proveden teoretický rozbor problematiky filtrace paketů. V kapitole 2 jsou charakterizovány aplikace z oblasti filtrace paketů v dnešních počítačových sítích. Hlavní důraz v navazujícím teoretickém rozboru je kladen na jeden z dílčích problémů filtrace – řešení problematiky klasifikace paketů, viz kapitola 3. Je zde definován problém klasifikace paketů (3.1) a jsou zde specifikovány obecné požadavky kladené na klasifikační algoritmy (3.2). V rámci dalších podkapitol jsou uvedeny základní přístupy ke klasifikaci paketů (3.3) a dva možné způsoby reprezentace problému klasifikace (3.4 a 3.5). Pro každý z nich jsou jednotlivě popsány a rozebrány z nich odvozené současné moderní metody provádění klasifikace paketů. Každá z uvedených metod je stručně charakterizována a shrnuta z pohledu časové a prostorové složitosti a také svých výhod a nevýhod vzhledem k ostatním přístupům. Kapitola 4 následně obsahuje výsledky experimentů a detailnější analýzy představených algoritmů na konkrétních sadách klasifikačních pravidel. Kapitola 5 se věnuje samotnému návrhu vlastní hardwarové architektury pro filtraci paketů ve 100 Gb sítích. Je zde postupně představen koncept obecné řetězené linky a dalšího přístupu v podobě oddělení klasifikace jednotlivých dimenzí. V rámci kapitoly 6 je popsána cílová hardwarová platforma COMBO použitá pro implementaci navrženého řešení. Jsou zde rozebrány možnosti akcelerační síťové karty COMBO-100G a vývojového prostředí NetCOPE. V kapitole 7 jsou na dostupných množinách filtračních pravidel prezentovány dosažené výsledky a vlastnosti nově navržené architektury a vytvořené implementace. Závěrečná kapitola 8 potom slouží jako shrnutí celkových výsledků práce a nabízí pohled na její možné budoucí pokračování.

Kapitola 2

Aplikace pro filtraci paketů

Následující kapitola se věnuje oblastem, ve kterých nachází filtrace paketů své uplatnění. V jednotlivých podčástech je charakterizována problematika filtrace síťového provozu využívající paketových filtrů a systémů IDS a IPS pro detekci a eliminaci nebezpečného síťového provozu. Poslední z oblastí, která je v rámci kapitoly uvedena, jsou zákonné odposlechy. Obsah následujícího textu vychází z literatury [1, 14].

2.1 Paketový filtr

Během posledních několika let se klade velký důraz na bezpečnost počítačových sítí. Případný útok musí být včas detekován a musí být včas zajištěna účinná ochrana počítačové sítě. Základním typem zařízení a nejrozšířenějším prvkem obrany proti útokům je paketový filtr, tzv. firewall. Firewally jsou síťová zařízení sloužící k eliminaci nežádoucího síťového provozu při komunikaci mezi počítačovými sítěmi. Obvykle se používají k řízení a kontrole příchozího a odchozího provozu na hranicích sítě, případně přímo na klientské stanici. Firewall umožňuje na základě definované sady pravidel povolit či zakázat procházející datový tok. Vývoj firewallů lze rozdělit do tří generací podle způsobu filtrace procházejícího datového toku:

Paketové filtry (stateless). Paketové filtry neuchovávají stav probíhajícího spojení. První generace firewallů provádí rozhodnutí o povolení či zakázání provozu pro každý paket komunikace zcela nezávisle (pouze na základě položek z hlaviček paketu a sady filtračních pravidel). Jejich hlavní výhodou je jednoduchost řešení. Na druhou stranu neumožňují rozpoznání zpětné komunikace u klientem iniciovaného odchozího spojení a problémy způsobuje i fragmentace paketů.

Stavové filtry (stateful). Stavové filtry řeší nedostatky předchozí generace firewallů a uchovávají nově i stav probíhající komunikace. Typickým příkladem je ustavení TCP spojení, při kterém je vložen záznam do stavové tabulky firewallu. Další pakety téže komunikace jsou už na základě záznamu automaticky povolovány. Díky uchování stavu lze rozlišovat směr komunikace a zajistit tak zabezpečení komunikace klientů z vnitřní sítě se servery vně.

Aplikační filtry (application proxy). Předchozí kategorie firewallů pracovaly nejvýše na síťové, případně transportní vrstvě. Tzv. proxy umožňují zkoumat i datovou část paketu až na úrovni aplikačního protokolu. Typickým použitím je například filtrace HTTP spojení na základě URL adresy.

V dnešní době se používají firewally především v podobě softwarové implementace a jsou zcela běžnou a nedílnou součástí operačních systémů. Existuje však i řada hardwarově orientovaných řešení. Jak již bylo řečeno, činnost firewallu je řízena pomocí množiny filtrovacích pravidel. Na základě porovnání informací z hlaviček paketu s množinou pravidel je daný paket propuštěn či nikoliv. Jednoduchý příklad možného formátu pravidel paketového filtru je uveden níže.

```
<NUMBER> <ACTION = accept|drop|reject>
  [proto <PROTOCOL>] [from <SRC-ADDRESS> [port <SRC-PORT>]]
  [to <DST-ADDRESS> [port <DST-PORT>]] [on <INTERFACE>]
```

Číslo pravidla *(NUMBER)* určuje jeho prioritu a pořadí vyhodnocování výsledné množiny pravidel. Následuje akce *(ACTION)*, která specifikuje způsob zpracování paketu, pokud vyhoví danému pravidlu. Paket může být přijat *(accept)*, zahozen *(drop)* nebo odmítnut *(reject)*. Dále jsou již uvedeny podmínky pro požadované hodnoty z hlaviček paketů, kterými jsou protokol transportní vrstvy *(PROTOCOL)*, zdrojová IP adresa *(SRC-ADDRESS)*, zdrojový port *(SRC-PORT)*, cílová IP adresa *(DST-ADDRESS)* a cílový port *(DST-PORT)*. Součástí filtračního pravidla může být dále například specifikace síťového rozhraní *(INTERFACE)*, přes které komunikace probíhá.

Ukázka sady filtračních pravidel a konfigurace paketového filtru *Netfilter* [13] integrovaného v jádře operačního systému *GNU/Linux* je uvedena níže.

```
Chain INPUT (policy DROP)
num action prot opt source          destination
1   ACCEPT tcp  --  0.0.0.0/0       0.0.0.0/0  state NEW tcp dpt:22
2   ACCEPT tcp  --  147.229.0.0/16  0.0.0.0/0  state NEW tcp dpt:80
3   ACCEPT tcp  --  147.229.0.0/16  0.0.0.0/0  state NEW tcp dpt:443
4   ACCEPT all  --  0.0.0.0/0       0.0.0.0/0  state RELATED,ESTABLISHED
5   DROP   icmp --  0.0.0.0/0       0.0.0.0/0
6   REJECT all  --  0.0.0.0/0       0.0.0.0/0  reject-with icmp-host-prohibited
```

Pravidla Netfilteru jsou organizována do řetězů, tzv. *chains*. Jedním z výchozích a vestavěných řetězů je *INPUT*, který řídí zpracování příchozích paketů určených hostitelskému počítači a který je uveden také v ukázce. Součástí konfigurace je také určení tzv. politiky, výchozí akce, která je provedena v případě, že paket nevyhoví žádnému z pravidel v daném řetězu. V našem případě je nastaveno zahazování paketů *(policy DROP)*. Vyhodnocení jednotlivých pravidel probíhá podle priority, vzestupně podle jejich číselného označení. Nejvyšší prioritu má pravidlo s číslem 1. První pravidlo povoluje ustavení příchozího SSH spojení (cílový port 22). Následující dvě pravidla dále povolují veškerou příchozí HTTP/HTTPS komunikaci (cílové porty 80 a 443) pocházející ze sítě VUT (podsíť 147.229.0.0/16). Pravidlo 4 propouští pakety všech již navázaných nebo s nimi souvisejících spojení. Pravidlo s číslem 5 zahazuje veškerou příchozí ICMP komunikaci. Veškerá další příchozí komunikace je odmítnuta aplikováním posledního z pravidel. Uvedená pravidla přitom kromě požadavků na hodnoty uvedené v hlavičkách paketů obsahují také specifikaci kontextu paketu v rámci ustavené relace – stav spojení. V rámci spojení se rozlišuje několik stavů: *NEW* (paket ustavující novou komunikaci), *ESTABLISHED* (pakety již ustaveného spojení), *RELATED* (pakety související s již probíhající komunikací) a *INVALID* (pakety, které nejsou součástí žádného z navázaných spojení). Stav spojení lze přitom při filtraci paketů chápat jako jeden z dalších vstupů klasifikačního algoritmu.

2.2 Systémy IDS a IPS

V dnešní době existuje řada útoků, které už z principu není možné pomocí paketového filtru odhalit. Tyto útoky totiž nelze jednoduše identifikovat pomocí informací z hlaviček paketů nebo stavu spojení. V řade případů se navíc jedná o útoky na standardní služby, které není možné prostřednictvím pravidel paketového filtru zcela zakázat. K detekování takových útoků je nutné využít systémů umožňujících provádění detailnější analýzy paketů.

Intrusion Detection System (IDS) je bezpečnostní systém, který provádí sledování síťového provozu, případně činnosti operačního systému. Jeho cílem je detekce a hlášení podezřelých aktivit představujících narušení zabezpečení sítě či systému. Na rozdíl od firewallů, které řeší bezpečnost sítě formou omezování přístupu a možnosti komunikace mezi sítěmi, je IDS pasivním systémem, který nijak nezasahuje do probíhajícího síťového provozu a umožňuje odhalit i útoky pocházející z vnitřní sítě.

Základem IDS je detailní analýza sledované síťové komunikace, nejen hlaviček paketů, ale i obsahu samotné komunikace (až na úrovni aplikačních protokolů). Principem detekce bezpečnostních incidentů je vyhledávání známých vzorů v komunikaci, například s využitím regulárních výrazů a jiných heuristických přístupů na základě vybudované databáze signatur. Systémy IDS tak umožňují velmi efektivní detekci známého škodlivého softwaru podle probíhající síťové komunikace. Podobným způsobem pracuje také většina antivirových programů. Značnou nevýhodou je však závislost na kvalitě a aktuálnosti používané databáze, možnost detekovat pouze známé hrozby a značné požadavky na výpočetní výkon. Známými implementacemi IDS jsou například Snort [20] nebo Bro [4].

Intrusion prevention system (IPS) je rozšířený systém IDS, který umožňuje kromě detekce podezřelé aktivity také proaktivně reagovat na zjištěnou událost a zabránit probíhajícímu útoku, případně minimalizovat způsobené škody. V takové situaci se například provádí zablokování zdroje nebezpečné síťové komunikace prostřednictvím automatizovaného nastavení firewallu. Systémy IPS lze považovat za pokročilou formu aplikačních firewallů.

Pravidla těchto systémů pro detekci nebezpečného síťového provozu se skládají ze dvou částí. Obsahují jednak definici položek hlaviček paketů, dále pak část popisující samotný obsah paketů. První část pravidla je popsána pomocí podmínek pro vybrané položky z hlaviček (až do úrovně transportní vrstvy), které musí být současně splněny. Typicky se jedná o následujících pět položek: Zdrojová IP adresa, cílová IP adresa, zdrojový a cílový TCP nebo UDP port a transportní protokol. Část pravidla věnovaná obsahu paketu se skládá z řetězců nebo je zadána ve formě regulárních výrazů, které mají být vyhledány v datech paketu nebo celém síťovém toku.

Příkladem zápisu konkrétního pravidla je pravidlo ze systému Snort uvedené níže, které detekuje pokus o přístup k síťovému souborovému systému NFS z vnější sítě.

```
alert tcp !192.168.1.0/24 any -> 192.168.1.0/24 111
(content: "|00 01 86 a5|"; msg: "external mountd access";)
```

Každé pravidlo se skládá z hlavičky a volitelného těla. Hlavička pravidla obsahuje nejprve definici akce *(alert)*, která se má provést, pokud zpracovávaný paket odpovídá pravidlu. Dále už následují podmínky pro vybrané položky z hlaviček paketu. Postupně je uveden typ transportního protokolu *(tcp)*, zdrojová IP adresa s maskou *(!192.168.1.0/24)*, zdrojový port *(any)*, cílová IP adresa s maskou *(192.168.1.0/24)* a cílový port *(111)*. Navazující tělo pravidla je uvedeno v kulatých závorkách. Jak je vidět na příkladu, tělo pravidla obsahuje podrobnější kritéria, která musí paket splňovat. Je zde možné definovat vzory nebo regulární výrazy, které se mají v datech paketu hledat. Pro každý z nich je dále možné specifikovat,

na jaké pozici v datech paketu se musí nacházet či jaká může být maximální vzdálenost mezi jednotlivými vzory. Kromě toho zde mohou být uvedeny také detailnější požadavky na nastavení některých položek v TCP/IP hlavičce. Jde například o pořadové číslo paketu nebo TCP příznaky. V těle pravidla se uvádí také text hlášení, které se má v případě úspěšné detekce vyvolat.

Požadavky pravidel vztažené k hlavičkám paketů vedou na řešení problému klasifikace paketů, tak jak jej definuje navazující kapitola 3.1 (a kterým se zabývá také tato diplomová práce). Řešení požadavků na vyhledávání řetězců a regulárních výrazů je již předmětem jiné oblasti označované jako pattern matching. Vyhodnocování pravidel u systémů IDS se tak skládá ze dvou částí. Provádí se analýza datové části paketu a následně klasifikace vybraných položek z hlaviček paketu, přičemž výstupy provedené analýzy datové části paketu jsou současně jedním ze vstupů klasifikačního algoritmu.

2.3 Zákonné odposlechy

Další z aplikací filtrace paketů jsou systémy pro zákonné odposlechy. Zákonné odposlechy (Lawful Interceptions, LI) reprezentují jeden z prostředků boje s kriminální činností v prostředí internetu. Základem je sledování aktivity a provádění záznamu a analýzy síťové komunikace podezřelých osob, které při trestné činnosti využívají prostředků veřejné počítačové, případně telefonní sítě. Celý proces i architektura podobných systémů je standardizována úřadem European Telecommunications Standards Institute (ETSI). Na základě soudního nařízení je odposlech fyzicky prováděn poskytovatelem telekomunikačních služeb (Internet Service Provider, ISP), který konkrétní získané informace dále poskytuje oprávněným orgánům činným v trestním řízení (Law Enforcement Agency, LEA).

Z hlediska technického zajištění odposlechu je nutné provádět identifikaci a filtraci zájmového provozu a ukládat pouze síťový provoz a komunikaci konkrétní osoby, pro kterou bylo vydáno příslušné povolení. Mezi shromažďované informace patří typicky nejen obsah samotné komunikace, ale i odpovídající metadata, například časové údaje nebo identifikace komunikujících stran.

Součástí požadavku na nový odposlech a vstupem do systému pro odposlechy je několik údajů: (1) Identifikace orgánu činného v trestním řízení (LEA), který odposlech požaduje, (2) jednoznačná a unikátní identifikace požadovaného odposlechu (Lawful Interception Identifier, LIID), (3) síťový identifikátor (Network Identifier, NID), který slouží k označení účastníků komunikace, případně konkrétního (např. TCP) spojení (typicky jde o statickou IP adresu nebo obecně rozsah adres definovaný adresou sítě a maskou), (4) datum a čas zahájení a ukončení odposlechu, (5) informace, zda je požadován pouze záznam metadat o komunikaci (Intercept Related Information, IRI) nebo kompletní síťový provoz odposlouchávaného subjektu (Content of Communication, CC). Z pohledu následné vlastní filtrace zájmového síťového provozu je přitom nejdůležitějším údajem síťový identifikátor (NID).

Filtrace paketů prováděná pro potřeby zákonných odposlechů má některá svá specifika. Jednou z důležitých vlastností je, že při filtraci paketů zájmového provozu je nutné síťový identifikátor porovnávat na shodu jak se zdrojem komunikace (zdrojovou IP adresou, případně zdrojovým portem), tak s cílem komunikace (cílovou IP adresou, případně cílovým portem). Za zájmový a následně zachytávaný provoz je pak považován každý paket, který splňuje alespoň jednu z těchto podmínek.

Kapitola 3

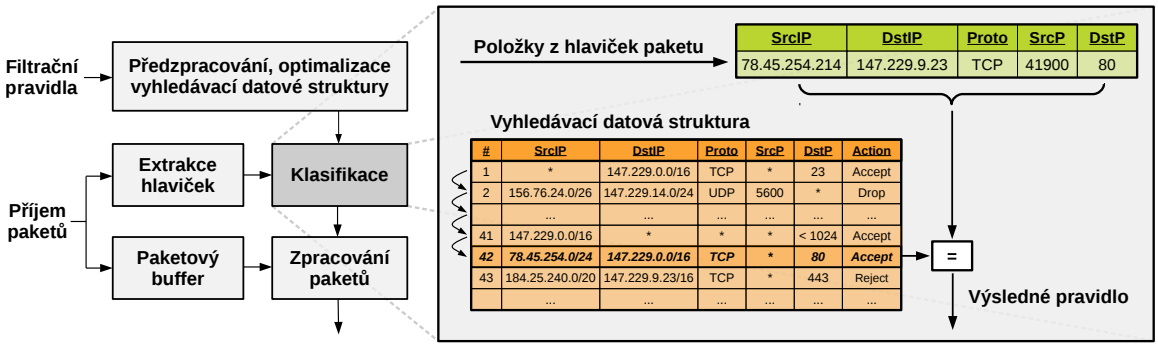
Klasifikace paketů

Tato kapitola je věnována problematice klasifikace paketů. V první části je uvedena definice problému a jsou představeny požadavky kladené na klasifikační algoritmy. Základní pohled na problém řešení klasifikace je poskytnut v následujících podkapitolách. Zbytek kapitoly se pak zabývá charakteristikou dalších sofistikovanějších metod řešení klasifikace. Není-li uvedeno jinak, popis problému klasifikace a jednotlivých přístupů k jeho řešení vychází z literatury [2, 12, 16].

3.1 Definice problému

Klasifikace paketů je problém, jehož cílem je roztřídit pakety podle zadaných kritérií – klasifikačních pravidel. Úkolem je ke každému příchozímu paketu co nejrychleji vyhledat pravidlo, jehož položky se shodují s hodnotami z hlaviček paketu. Vstupem algoritmu je množina klasifikačních pravidel uspořádaná podle priority a hodnoty extrahované z hlaviček klasifikovaného paketu. Výsledkem je pak pravidlo odpovídající danému paketu. V případě, že paketu odpovídá více pravidel, je vybráno to s nejvyšší prioritou.

Proces klasifikace paketů je ilustrován na obrázku 3.1. Vlevo na obrázku je nejprve znázorněno zasazení klasifikace do širšího kontextu. Při filtraci jsou příchozí pakety ukládány do paketového bufferu. Zároveň probíhá extrakce jejich hlaviček. Podle informací z hlaviček je potom v rámci procesu klasifikace nalezeno odpovídající pravidlo, podle něhož je paket uložený v bufferu dále zpracován. V pravé části obrázku je detailněji znázorněn samotný průběh klasifikace. Zeleně jsou znázorněny extrahované položky z hlaviček paketu.



Obrázek 3.1: *Problém klasifikace a zasazení klasifikace do širšího kontextu.*

Oranžová tabulka potom představuje datovou strukturu optimalizovanou pro vyhledávání pravidel. Výsledným pravidlem je potom z pohledu priority první pravidlo, které odpovídá klasifikovanému paketu. Na příkladu z obrázku je to pravidlo s číslem 42.

Jednotlivé položky hlaviček klasifikovaného paketu H (header) tvoří k-tici ($H_1, H_2, \ldots, H_k$), kde k je počet polí a H_i obsahuje sekvenci bitů daného pole. Pouze některé z položek jsou významné a používají se pro potřeby klasifikace. Jsou označovány jako tzv. dimenze. Typicky jde o pětici tvořenou zdrojovou a cílovou IP adresou (32 nebo 128 bitů), zdrojovým a cílovým portem (16 bitů) a typem transportního protokolu (8 bitů). U dimenze uvažujeme její šířku, tzn. počet bitů daného pole.

Klasifikační pravidlo R (rule) tvoří $(k+2)$-tici $(p, c_1, c_2, \ldots, c_k, a)$, kde k je počet dimenzí klasifikátoru, $p \in \mathbb{N}$ je priorita pravidla, c_i jsou podmínky definované pro každou dimenzi a a je akce přidružená k pravidlu. Klasifikovaný paket H vyhohuje pravidlu R_i, pokud všechna pole H_j hlaviček paketu odpovídají podmínkám $R_i(j)$ daného pravidla pro $1 \leq j \leq k$. Jelikož klasifikovaný paket může vyhovovat více pravidlům (pravidla se mohou obecně překrývat), definuje navíc každé pravidlo svou prioritu. Priorita je přirozené číslo, kde nižší hodnota představuje vyšší prioritu pravidla. Přidružená akce pak specifikuje způsob zpracování paketu po jeho klasifikaci. Každé pravidlo R_i určuje také typ porovnávání hodnoty $R_i(j)$ pro každou z dimenzí k. Podmínky porovnávání hodnot v jednotlivých dimenzích mohou být následující:

Intervalové porovnání (range match). Pro intervalové porovnávání musí být hodnota pole paketu H_j v rozsahu zadaném klasifikačním pravidlem $R_i(j)$. Podmínka tak určuje souvislý rozsah hodnot vyhovujících klasifikačnímu pravidlu. Tento typ podmínky se typicky používá pro omezení čísla TCP, respektive UDP portu.

Shoda prefixu (prefix match). U prefixového porovnání musí být hodnota pravidla $R_i(j)$ prefixem hodnoty pole paketu H_j. Pro porovnání je zadáno pouze několik vyšších bitů datového slova. Nižší bity mohou mít libovolnou hodnotu. Tyto podmínky se používají pro definici skupiny IP adres, kde prefix odpovídá adrese sítě, přičemž konkrétní adresa koncového zařízení může být libovolná a odpovídá tak IP adrese libovolné stanice v dané síti. Prefixové porovnání je zároveň speciálním případem rozsahu.

Přesné porovnání (exact match). Při přesném porovnání musí mít pole paketu H_j přesně definovanou hodnotu odpovídající položce v klasifikačním pravidle $R_i(j)$. Jedná se o speciální případ prefixu i rozsahu. Tento typ podmínky se používá pro určení typu protokolu (TCP, UDP, ICMP), případně pro příznaky protokolu TCP (hodnoty SYN, ACK, FIN nebo RST).

Libovolná hodnota (any match). Pole paketu H_j může mít zcela libovolnou hodnotu. Jedná se o speciální případ prefixového i intervalového porovnání.

Regulární výraz (regular expression match). Porovnání s využitím regulárních výrazů se používá při analýze obsahu paketů, například pro vyhledávání řetězců na úrovni aplikačních protokolů (HTTP, SMTP). Tento typ porovnání je především u vysokorychlostních sítí velmi náročný na zpracování.

Zatímco podmínkami pravidel a porovnáními vztaženými k hlavičkám paketů se zabývá přímo oblast klasifikace paketů, vyhledávání řetězců a regulárních výrazů v datové části paketu už spadá do jiné oblasti označované jako pattern matching, která využívá jiných přístupů k řešení a kterou se diplomová práce dále nezabývá.

Všechny typy porovnání klasifikačního pravidla (mimo již zmíněné regulární výrazy) jsou speciálním případem rozsahu. Každý rozsah lze ale převést také na sadu několika prefixů. Libovolnou hodnotu lze chápat jako prefix, kde jsou všechny bity volitelné (prefix nulové délky). Přesné porovnání lze zase chápat jako prefix, kde žádné bity nejsou volitelné (prefix maximální délky). Nejhorším možným příkladem převodu rozsahu na sadu prefixů je interval $\langle 1, 2^N - 2\rangle$, kde N je počet bitů daného pole. Při tomto převodu vzniká až $(2N - 2)$ různých prefixů (nových pravidel). Pravidla, která vzniknou převodem rozsahů na prefixy mohou mít přiřazenu stejnou prioritu, neboť výsledné prefixy se nikdy nepřekrývají.

Klasifikátor potom provádí prohledání konečné množiny n klasifikačních pravidel (rules) $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. Výsledek procesu klasifikace přitom odpovídá lineárnímu průchodu pravidel seřazených do neklesající posloupnosti podle priority a výběru prvního vhodného pravidla. Klasifikaci paketů lze dále rozdělit do dvou fází – předzpracování množiny klasifikačních pravidel $\mathcal{R}$ a vlastní klasifikace příchozích paketů.

Fáze předzpracování. Cílem předzpracování je příprava klasifikačních pravidel $\mathcal{R}$ a jejich uspořádání do vhodné datové struktury optimalizované pro vyhledávání. Předzpracování je prováděno v okamžiku přidávání, modifikace nebo rušení pravidla. Tato operace se provádí méně často a není proto tak časově kritická.

Fáze klasifikace. Při samotné fázi klasifikace dochází ke zpracování příchozího síťového provozu, extrakci polí z hlaviček H paketů. Následně je procházena datová struktura, jež byla vytvořena v rámci prvního kroku. Hodnoty H_j z hlaviček paketů jsou porovnávány s podmínkami $R_i(j)$ z pravidel a je vyhledáno odpovídající klasifikační pravidlo s nejvyšší prioritou. Operace je kritická, neboť požadujeme co nejrychlejší zpracování paketu, aby nebyla omezena propustnost daného síťového zařízení.

3.2 Požadavky na klasifikační algoritmy

Pro potřeby klasifikace paketů hledáme vhodné algoritmy, které budou na jedné straně dostatečně obecné, rozšiřitelné a budou podporovat vyhledávání ve více dimenzích. Na druhé straně požadujeme, aby byly rychlé, efektivní a jejich časová a paměťová složitost umožňovala implementaci na dostupných hardwarových platformách. Typicky hledáme kompromis v algoritmu splňujícím následující vlastnosti:

Rychlost klasifikace. Jedním z hlavních kritérií hodnocení algoritmů je požadavek na výkonnost – rychlost klasifikace. Klasifikátor musí být schopen pracovat v reálném čase a nesmí při tom negativně ovlivňovat propustnost celého zařízení. Nejhorším případem je zpracování nejkratších paketů, neboť to odpovídá největšímu počtu prováděných klasifikací za jednotku času. Například pro síť typu Ethernet s nejmenším přenášeným rámcem o velikosti 64 B a s přenosovou rychlostí 100 Gb/s to odpovídá 148 809 523 zpracovaným paketům a provedeným klasifikacím za sekundu.

Rozhodujícím faktorem ovlivňujícím rychlost klasifikace může být typ použité paměti (DRAM, SRAM) a s tím související přístupová doba do paměti. Z praktického hlediska se tak složitost klasifikačního algoritmu často uvádí jako počet přístupů do externí paměti potřebný při klasifikaci jediného paketu. Další metrikou může být reálná propustnost v paketech za sekundu, doba mezi dvěma vyhledáními nebo latence klasifikace.

Paměťová náročnost. Významným faktorem je velikost použité paměti, kterou potřebujeme pro uložení vyhledávací struktury, neboť ta zásadním způsobem ovlivňuje nejen rychlost, ale i výslednou cenu zařízení. Od paměťové náročnosti se totiž odvíjí typ použité paměti. Menší paměti (SRAM), dostupné na čipu, jsou mnohem rychlejší než externí paměti s vyšší kapacitou (DRAM). Velikost potřebné paměti je typicky udávána v počtu bajtů na pravidlo.

Rychlost aktualizace struktury. Některé aplikace pro filtraci paketů vyžadují provádění změn množiny klasifikačních pravidel. V některých případech je potřeba rychlá, někdy málo častá změna. Pro určité aplikace tak může být velmi užitečné, pokud daný algoritmus podporuje dynamické změny množiny pravidel včetně řešení atomicity a konzistence operací. Umožňuje tak provádět změny za běhu bez nutnosti pozastavení procesu klasifikace. Způsob provádění aktualizace množiny pravidel závisí na použité datové struktuře. Některé přístupy vyžadují při jakékoliv modifikaci množiny pravidel znovuvytvoření celé datové struktury. Tato operace je ale časově i prostorově náročná.

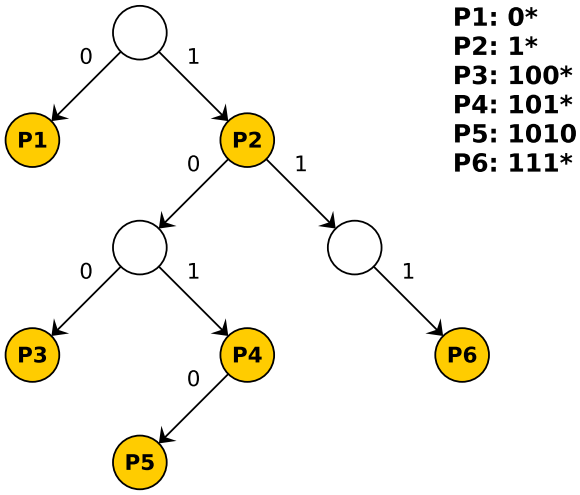
Počet a vlastnosti pravidel. Významnou vlastností algoritmu je jeho rozšiřitelnost. Přidáním dalšího pravidla nesmí dojít k výraznému poklesu výkonnosti systému nebo naopak nárůstu využité paměti. Existuje proto omezení na maximální počet pravidel, které lze do systému nahrát, aby bylo zachováno efektivní zpracování. Důležitou vlastností algoritmu je také počet dimenzí, tzn. počet polí z hlaviček paketů, které je schopen zpracovat. Lze tak rozlišovat jedno-, dvou- a vícedimenzionální metody. Ideálně není algoritmus na počtu dimenzí závislý a je schopen zpracovat jakýkoliv počet polí. Omezení může být kladeno i na unikátní počet hodnot v jednotlivých dimenzích. Ten je většinou významně nižší než celkový počet pravidel klasifikátoru. Toho potom využívají pokročilé metody klasifikace.

Složitost implementace. Složitost implementace vyjadřuje, zda bude možné algoritmus implementovat na určité hardwarově specifické platformě a případně jaké zdroje bude implementace zabírat. Míru složitosti lze v případě hardwarové implementace vyjádřit například jako zabranou plochu na čipu (FPGA či ASIC).

3.3 Základní přístupy ke klasifikaci

Nejjednodušším přístupem k problému klasifikace je již dříve naznačený lineární průchod množiny pravidel. Všechna klasifikační pravidla jsou uložena do sekvenční datové struktury a při klasifikaci je každý paket postupně porovnáván se všemi pravidly klasifikátoru. Tento algoritmus má zřejmě nejmenší nároky na paměť. Jde však o zcela nevhodný přístup z pohledu rychlosti zpracování a počtu potřebných přístupů do paměti. Časová složitost algoritmu je lineární $\mathcal{O}(n)$, kde n je počet klasifikačních pravidel. Pro ukládání pravidel a vyhledávání se používají daleko efektivnější datové struktury.

Dalším z naivních algoritmů klasifikace je zcela ortogonální přístup, kdy jsou všechna pole H_j z hlaviček paketu konkatenována do jediného širokého datového slova, které následně slouží jako index do předpočítané tabulky v paměti. Na určené adrese v paměti je pak uložen identifikátor výsledného pravidla. Časová složitost tohoto přístupu je konstantní $\mathcal{O}(1)$ a k určení výsledku klasifikace stačí jediný přístup do paměti, čímž je takový algoritmus z tohoto pohledu teoreticky nejlepší možný. Reálnému použití algoritmu ale brání



Obrázek 3.2: *Ukázka binární struktury trie pro čtyřbitové pole pro vyhledávání prefixů.*

velmi vysoká paměťová náročnost, kdy je třeba mít v paměti uloženo číslo pravidla pro všechny možné pakety.

3.3.1 Trie a LPM

Prefixové vyjádření pravidel je výhodné z důvodu přímočaré stromové reprezentace. Nejjednodušší binární stromová datová struktura pro efektivní uložení sady prefixů se nazývá trie, pojmenovaná podle slova retrieval (vyhledání). S prefixovou reprezentací souvisí také operace hledání nejdelšího shodné prefixu (Longest Prefix Match, LPM), jejímž vstupem je sada prefixů různé délky a konkrétní klasifikovaná hodnota. Výstupem operace je pak ze všech vstupních prefixů ten, který odpovídá dané hodnotě a je nejdelší (nejspecifičtější). LPM je příkladem jednodimenzionální klasifikace používané při vyhledávání ve směrovacích tabulkách.

Příklad struktury trie je uveden na obrázku 3.2. Hrany stromu jsou ohodnoceny bity 0 nebo 1 a každý z vnitřních uzlů stromové reprezentace obsahuje ukazatel na jednoho nebo dva potomky. Jednotlivé prefixy jsou uloženy přímo v konstrukci stromu a odpovídají některým uzlům stromu, na obrázku 3.2 jsou barevně zvýrazněny a označeny číslem prefixu. Prefix reprezentovaný uzlem u je dán konkatenací ohodnocení hran na cestě od kořene stromu k uzlu u, kde hrany odpovídají jednotlivým bitům prefixu. Například uzel P4 reprezentuje prefix 101*. Listové uzly nemají žádné potomky a vždy reprezentují některý z prefixů. Výška stromu pak odpovídá délce nejdelšího prefixu.

Při vyhledávání zpracováváme postupně bity hledaného slova od nejvýznamnějšího (most significant bit, MSB) směrem k méně významným bitům a souběžně procházíme strom od kořene k listům. Podle zpracovávaného bitu v každém kroku výpočtu rozhodujeme, zda budeme dále postupovat levým nebo pravým podstromem. Při průchodu zaznamenáváme vždy poslední (dosud nejdelší) navštívený prefix a postupujeme tak dlouho, dokud existuje větev odpovídající svým ohodnocením zpracovávanému bitu. V případě, že dorazíme do uzlu, v němž neexistuje požadovaná větev, vrátíme poslední uložený prefix, který je současně nejdelším shodným prefixem pro hledanou hodnotu. Časová složitost vyhledání je lineární $\mathcal{O}(W)$, kde W je výška stromu a odpovídá počtu bitů datového slova.

Nevýhodou binárních stromů je sekvenční porovnávání každého jednotlivého bitu datového slova. To může být příliš pomalé a vždy představuje přístup do paměti. Proto přichází řada modifikací základního přístupu s myšlenkou vícebitových stromů trie, které umožňují porovnat více bitů v jediném kroku. Tím se úměrně snižuje i počet potřebných přístupů do paměti.

3.3.2 TCAM

Jedním z dalších základních přístupů je použití ternární asociativní paměti (TCAM). TCAM je speciálním typem asociativní paměti, která umožňuje vyhledávání podle obsahu. Paměť je organizována do řádků složených z elementárních buněk, které odpovídají jednotlivým bitům hledaného slova. Vstupem je tedy hledané datové slovo, výstupem pak adresa (řádek), kde se slovo nachází.

Každá buňka paměti obsahuje bitový komparátor vyhodnocující shodu uloženého slova se slovem hledaným. Řádek paměti je výsledkem hledání, pokud binární stavy buněk v řádku svou hodnotou odpovídají jednotlivým bitům hledaného slova. Ternární varianta paměti navíc pracuje kromě dvou binárních stavů (jedničky a nuly) se třetím stavem X (don't care). Při porovnávání je taková bitová pozice ignorována a daný bit je vždy považován za shodný. Nastavením horní části bitů datového slova na požadovanou hodnotu a označením zbylých bitům příznakem don't care tak TCAM přímo umožňuje vyhledávání prefixů.

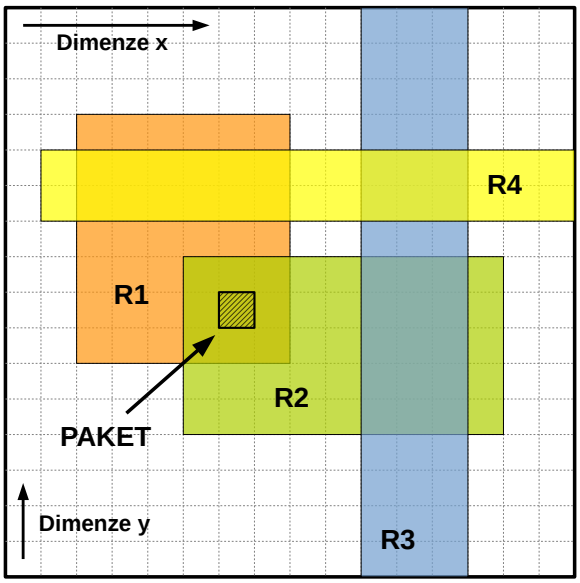
Řádek paměti lze tímto způsobem využít k uložení jednoho klasifikačního pravidla vyjádřeného ve formě prefixů. Při klasifikaci je z položek hlaviček paketu vytvořeno jediné široké datové slovo, které je použito k vyhledání v paměti. Vyhledávání je možné jediným přístupem do paměti. Časovou složitost klasifikace však nelze považovat za konstantní $\mathcal{O}(1)$, neboť ve skutečnosti je prováděno n paralelních hledání pro každý řádek paměti, kde n je počet klasifikačních pravidel.

Výhodou TCAM je rychlost vyhledání a jednoduchost použití. Na druhou stranu mají omezenou kapacitu, velký příkon a vysokou pořizovací cenu vztaženou na bit v porovnání s běžným typem pamětí (SRAM). Vzhledem k uvedeným vlastnostem jsou TCAM používány spíše jako malé vyrovnávací paměti k optimalizaci jiných přístupů ke klasifikaci.

3.4 Geometrický přístup ke klasifikaci

K problému klasifikace lze přistupovat jako ke geometrickému problému, kde každé pole z hlavičky paketu představuje jednu z dimenzí diskrétního vícedimenzionálního prostoru. Klasifikační pravidla v této reprezentaci dostávají podobu pravoúhlých objektů v tomto prostoru, kde každá z podmínek pravidla definuje rozsah hodnot v určité dimenzi. Tyto objekty (pravidla) se mohou v prostoru libovolně překrývat.

Geometrická reprezentace problému klasifikace je ilustrována na obrázku 3.3. Obrázek znázorňuje dvoudimenzionální prostor, kde osy x a y představují jednotlivé dimenze. Pravidla jsou potom reprezentována barevnými obdélníky (R_1 červeným, R_2 zeleným, R_3 modrým a R_4 žlutým obdélníkem), jež se v prostoru určitým způsobem překrývají. Každý klasifikovaný paket pak určuje v prostoru bod, který může být současně obsažen v jednom nebo i více objektech reprezentujících klasifikační pravidla. Úkolem procesu klasifikace je potom z objektů (pravidel), které obsahují bod reprezentující daný paket (odpovídají danému paketu), vybrat ten s největší prioritou. Z geometrické reprezentace vychází následující algoritmy založené na dělení prostoru a konstrukci rozhodovacích stromů.



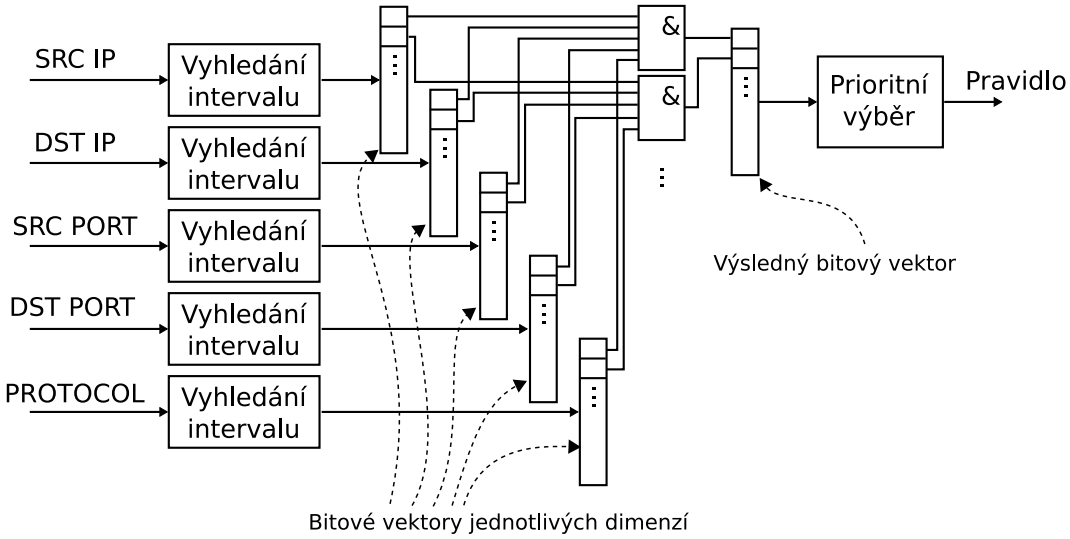
Obrázek 3.3: *Geometrické reprezentace problému klasifikace paketů.*

3.4.1 BitVector

Algoritmus BitVector [9] je první ze skupiny algoritmů vycházejících z geometrické reprezentace problému klasifikace. Klasifikační pravidla jsou vyjádřena ve formě rozsahů. V rámci fáze předzpracování je provedeno zobrazení rozsahů na číselnou osu každé dimenze. Tím vznikne v každé dimenzi maximálně $(2n+1)$ disjunktních intervalů, kde n je původní počet pravidel. Ke každému takovému intervalu je pak přiřazen bitový vektor o délce n, ve kterém jednotlivé bity odpovídají klasifikačním pravidlům. Bity vektoru jsou nastaveny tak, že pokud se dané pravidlo překrývá se zpracovávaným intervalem, je na odpovídající bitové pozici hodnota 1. V opačném případě jsou uvedeny nuly. Klasifikační pravidla jsou seřazena podle priority, takže první bit vektoru odpovídá pravidlu s nejvyšší prioritou.

Struktura algoritmu je zobrazena na obrázku 3.4. Při samotné klasifikaci se pro extrahované hodnoty z hlaviček paketu (*SRC-IP*, *DST-IP*, *SRC-PORT*, *DST-PORT* a *PROTOCOL*, viz obrázek 3.4) provádí vyhledání příslušného intervalu v každé dimenzi (na obrázku jako bloky *vyhledání intervalu*). Tento krok lze provést s logaritmickou časovou složitostí například pomocí binárního vyhledávání. Pro každou dimenzi získáme odpovídající bitový vektor. Hledání ve všech dimenzích lze přitom provádět paralelně. Nad všemi vektory se následně provede operace bitového součinu (bloky se symbolem &). Výsledkem je jediný vektor, jehož bitové pozice odpovídají pravidlům, která jsou splněna ve všech dimenzích. Výběrem pravidla, které přísluší nejnižší bitové pozici s hodnotou jedna, získáme pravidlo s nejvyšší prioritou, které odpovídá klasifikovanému paketu.

Délka vektoru odpovídá počtu klasifikačních pravidel. Algoritmus BitVector má tak časovou složitost vyhledání $\mathcal{O}(n)$, kde n je velikost bitového vektoru. Nevýhodou algoritmu je právě lineární závislost velikosti vektoru na počtu pravidel. Pro vyšší počty pravidel a větší velikost vektoru je totiž potřeba vícenásobný přístup do paměti pro jeho načtení. Některé z modifikací základního algoritmu, například Aggregated BitVector nebo BV-TCAM, zlepšují časovou i prostorovou náročnost algoritmu efektivnějším uložením bitových vektorů v paměti, případně kombinací algoritmu s pamětmi TCAM.



Obrázek 3.4: *Základní struktura algoritmu BitVector (vyhledání intervalu v každé dimenzi, bitový součin, výběr pravidla s nejvyšší prioritou); [2].*

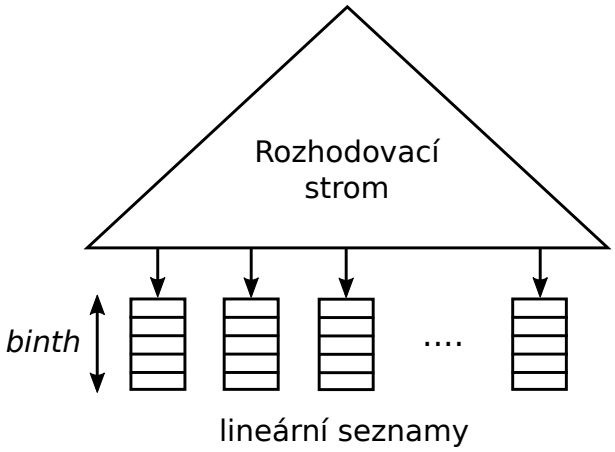
3.4.2 HiCuts a HyperCuts

Oba algoritmy HiCuts [8] i HyperCuts [19] vychází z geometrické reprezentace problému a jsou založeny na konstrukci rozhodovacího stromu. Ve fázi předzpracování je v rámci vnitřních uzlů konstruovaného stromu prohledávaný prostor dělen pomocí rovinných ploch. Vznikající podprostory stejné velikosti jsou dále reprezentovány jako potomci zpracovávaného uzlu. Výběr dělené dimenze a počet prováděných řezů (počet dělení) je určen heuristickou funkcí v závislosti na struktuře klasifikačních pravidel a vstupních parametrech algoritmu. Při konstrukci stromu každý z uzlů obsahuje informaci o rozsahu, který reprezentuje, podmnožině pravidel, které odpovídá, dělené dimenzi a počtu řezů. Dělení je prováděno rekurzivně, dokud počet pravidel v daném podprostoru neklesne pod stanovenou hranici, parametr *binth*. Listové uzly pak obsahují ukazatel na lineární seznam zbývajících pravidel. Obrázek 3.5 ilustruje stromovou datovou strukturu používanou algoritmem HiCuts. Na obrázku je znázorněn rozhodovací strom, přičemž poslední úroveň stromu (listové uzly) obsahuje lineární seznamy pro uložení až *binth* pravidel.

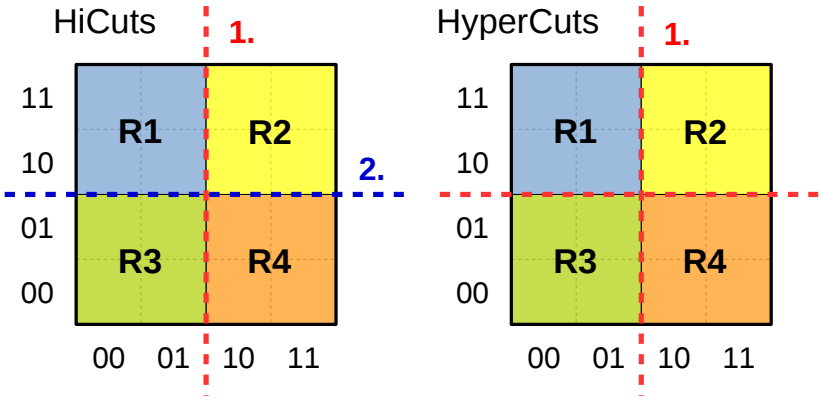
Při samotné klasifikaci je vytvořený rozhodovací strom procházen od kořene k listům a v každém kroku je vybrán takový potomek uzlu, který reprezentuje podprostor, ve kterém se nachází klasifikovaný paket. Prohledávaný prostor je tímto způsobem omezen tak, že procházíme pouze oblast obsahující správná pravidla. V rámci uzlů na listové úrovni je pak sekvenčním průchodem lineárního seznamu dohledáno nejlepší pravidlo odpovídající klasifikovanému paketu.

Časová složitost vyhledání i spotřeba paměti závisí na vnitřní struktuře zkonstruovaného stromu (počtu prováděných dělení a počtu potomků každého uzlu). Stromy s větším počtem řezů v každém uzlu sníží celkovou výšku stromu a umožní rychlejší vyhledání, na druhou stranu však zvyšují paměťové nároky. Volbou parametrů algoritmu lze však nalézt vhodné rozložení mezi časovou a prostorovou složitostí.

Přístup HyperCuts je modifikací základního algoritmu HiCuts, který zavádí možnost dělení prostoru v jednom kroku podle více dimenzí. Tím umožňuje snížení výšky stromu a časové i paměťové náročnosti. Srovnání přístupů obou metod ilustruje obrázek 3.6. Na



Obrázek 3.5: *Stromová datová struktura používaná algoritmem HiCuts (listové uzly umožňují v rámci lineárního seznamu uložení až* binth *pravidel).*



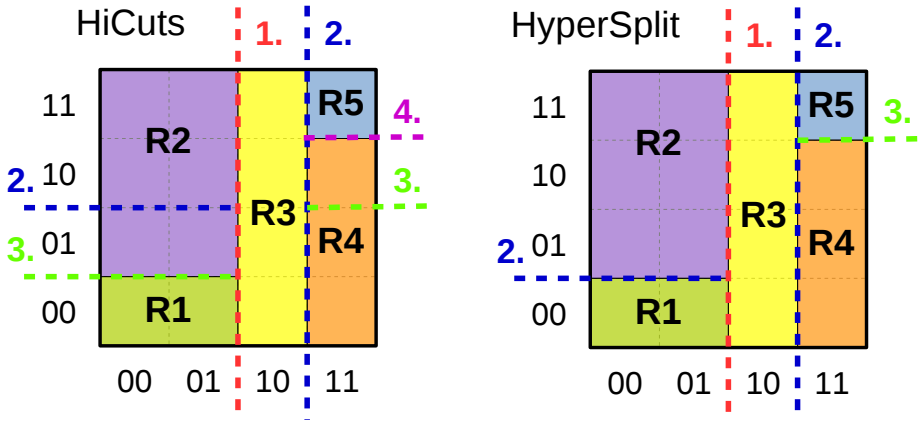
Obrázek 3.6: *Srovnání přístupu metod HiCuts a HyperCuts.*

obrázku je znázorněn dvoudimenzionální prostor pokrytý pravidly R_1 (modře), R_2 (žlutě), R_3 (zeleně) a R_4 (oranžově). Barevně, přerušovanou čarou, jsou naznačeny řezy prostoru. Zatímco HiCuts (vlevo) potřebuje k rozdělení prostoru klasifikačních pravidel z obrázku dva kroky, HyperCuts (vpravo) umožňuje díky práci s více dimenzemi rozdělení prostoru v jediném kroku.

3.4.3 HyperSplit

Algoritmus HyperSplit [18, 26] je jednou z dalších modifikací základního přístupu metody HiCuts. Na rozdíl od HiCuts a HyperCuts dovoluje HyperSplit při konstrukci rozhodovacího stromu dělení prostoru pouze na dvě části a umožňuje také dělení vždy jen jedné dimenze současně. Při předzpracování sady klasifikačních pravidel tak pokaždé vzniká binární rozhodovací strom. Nejdůležitější odlišností od základního přístupu je však možnost nerovnoměrného dělení prostoru. To je možné díky ukládání hraniční hodnoty v uzlech stromu a použití intervalového (nikoliv prefixového) porovnávání hodnot při klasifikaci.

Na obrázku 3.7 jsou ilustrovány rozdíly v přístupu k dělení prostoru u metod HiCuts a HyperSplit. Na složitějším příkladu sady klasifikačních pravidel z obrázku se ukazuje, že



Obrázek 3.7: *Srovnání přístupu k dělení prostoru u metod HiCuts a HyperSplit*

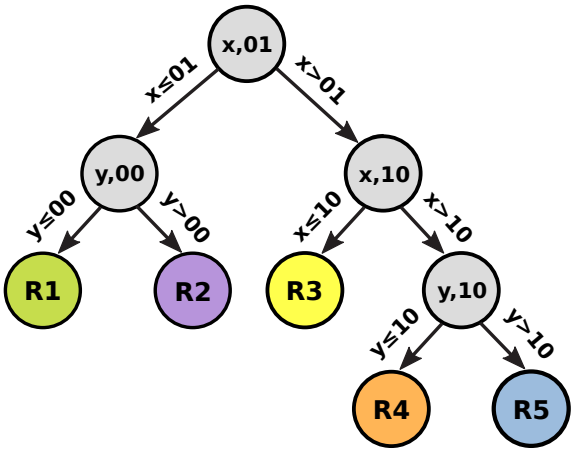
algoritmus HiCuts (vlevo) vyžaduje celkově čtyři kroky k definitivnímu rozdělení prostoru na jednotlivá pravidla. Algoritmus HyperSplit (vpravo) ale díky možnosti nerovnoměrného dělení prostoru umožňuje řešení problému efektivněji – pouze ve třech krocích. Obrázek 3.8 ukazuje také odpovídající binární rozhodovací strom zkonstruovaný přístupem HyperSplit. V rámci každého vnitřního uzlu stromu je uložena jednak hraniční hodnota a dále také dimenze, podle které je prováděn řez. Každý listový uzel potom obsahuje odpovídající pravidlo (R_1 až R_5).

Metoda HyperSplit uvažuje vyjádření klasifikačních pravidel ve formě rozsahů. Při konstrukci rozhodovacího stromu se postupuje rekurzivně. Pro zpracovávaný (pod)prostor a odpovídající (pod)množinu klasifikačních pravidel je provedeno zobrazení rozsahů na číselnou osu každé dimenze. Tím vznikne v každé dimenzi M hraničních hodnot, kde každé dvě po sobě jdoucí hodnoty oddělují na číselné ose jeden z $(M-1)$ disjunktních segmentů. Pro počet hraničních hodnot rovněž platí $2 \leq M \leq (2N+1)$, kde N je původní počet pravidel. Při výběru hraniční hodnoty vhodné k provedení řezu dimenze se používá několik strategií (heuristik). Pro následující vysvětlení předpokládejme, že P_t je pole hraničních hodnot seřazených vzestupně, $P_t[i]$ značí i-tou z nich, kde $1 \leq i \leq M$, a $S_g[j]$ označuje j-tý segment, kde $1 \leq j \leq (M-1)$.

První strategie (segment-balanced). Pro řez dimenze je vybrána hodnota $P_t[\lfloor M/2 \rfloor]$. Tato strategie provádí dělení prostoru s ohledem na dosažení vyváženého počtu segmentů v nově vznikajících podprostorech.

Druhá strategie (rule-balanced). Pro řez dimenze je vybrána taková hodnota $P_t[m]$, že počet pravidel v intervalu $\langle P_t[1], P_t[m] \rangle$ odpovídá přibližně polovině zpracovávaného počtu klasifikačních pravidel, tzn. hodnotě $\lfloor N/2 \rfloor$. Tato strategie při dělení prostoru provádí vyvážení počtu pravidel ve vznikajících podprostorech.

Třetí strategie (weighted segment-balanced). Předpokládejme, že $S_r[j]$ značí počet pravidel, které se překrývají se segmentem $S_g[j]$. Pro řez dimenze je pak vybrána taková hodnota $P_t[m]$, kde m je minimální a takové, že splňuje nerovnost $\sum_{j=1}^{m} S_r[j] > \frac{1}{2} \sum_{j=1}^{M} S_r[j]$. Tato strategie umožňuje trochu odlišný způsob vyvážení počtu segmentů ve vznikajících podprostorech. Na rozdíl od prvního přístupu přidává každému segmentu $S_g[j]$ váhu, která odpovídá počtu pravidel $S_r[j]$ překrývajících se s daným segmentem, a zohledňuje tak vnitřní strukturu vznikajících podprostorů.



Obrázek 3.8: *Rozhodovací strom zkonstruovaný přístupem HyperSplit a odpovídající příkladu z obrázku 3.7.*

Dalším z důležitých faktorů při konstrukci rozhodovacího stromu je výběr vhodné dimenze k provedení řezu. Pro první dvě strategie je možné vybírat vždy takovou dimenzi, která obsahuje největší počet $(M-1)$ segmentů. U třetí strategie je každému segmentu přiřazována váha. Je tedy možné zase vybírat takovou dimenzi, pro kterou je průměrná váha segmentu nejmenší, tzn. takovou, pro kterou je hodnota $\frac{1}{M}\sum_{j=1}^{M} S_r[j]$ minimální.

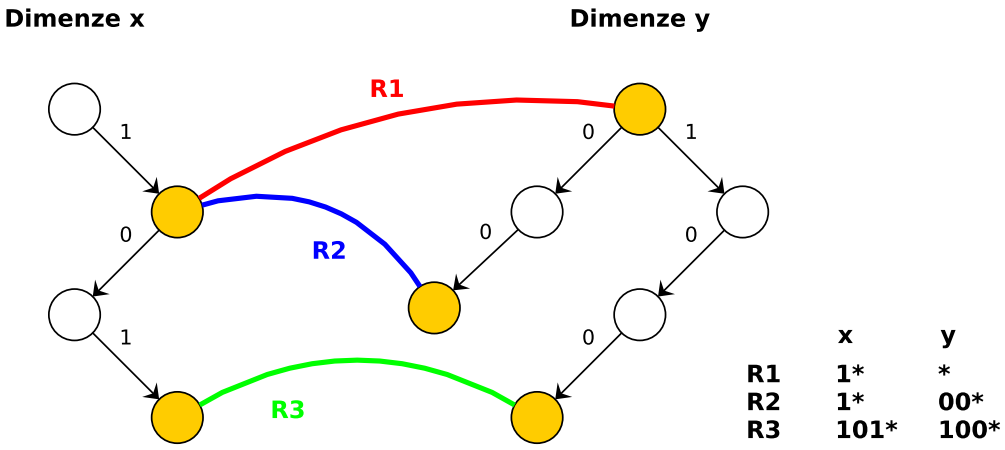
V každém stupni rekurze je vždy nově vzniklému podprostoru přiřazena podmnožina zpracovávané sady pravidel, která se s daným podprostorem překrývá. Ukončení rekurzivního dělení prostoru je potom možné na základě jedné ze dvou podmínek. První možností je, že počet pravidel v daném podprostoru podobně jako u algoritmu HiCuts klesne pod předem stanovenou hranici, parametr *binth*. Pro $binth > 1$ je potom výsledné pravidlo dohledáno sekvenčně v rámci souvisejícího lineárního seznamu. Druhou možností je, že aktuálně zpracovávaný podprostor je plně pokryt všemi pravidly odpovídající podmnožiny. V tomto případě je výsledkem jediné pravidlo, to s nejvyšší prioritou.

Nerovnoměrné dělení prostoru, které provádí algoritmus HyperSplit, vede na celkově lepší vyváženost a efektivitu konstruovaného rozhodovacího stromu a na rozdíl od algoritmů HiCuts a HyperCuts tak umožňuje významnou redukci paměťové i časové složitosti.

3.5 Kombinatorický přístup ke klasifikaci

Na klasifikaci lze pohlížet také jako na kombinatorický problém. Každá dimenze je potom charakterizována množinou prefixů obecné délky, přičemž některé z delších prefixů dané dimenze mohou být zcela obsaženy v ostatních, kratších prefixech. V rámci každé dimenze lze množinu prefixů vyjádřit binárním stromem trie. Klasifikační pravidla v této reprezentaci mají potom podobu spojnice uzlů (vždy jednoho uzlu z každého binárního stromu reprezentujícího jednu dimenzi). Kombinatorickou reprezentaci problému klasifikace ilustruje obrázek 3.9. Na obrázku je zobrazen příklad sady tří pravidel R_1, R_2 a R_3 se dvěma dimenzemi. Prefixy z každé dimenze jsou vyjádřeny binárními stromy. Spojnice reprezentující klasifikační pravidla jsou na obrázku znázorněna barevně, pravidlo R_1 červeně, R_2 modře a R_3 zeleně.

Prvním krokem procesu klasifikace je vyhledání prefixů, které odpovídají klasifikovanému paketu, zvlášť v každé dimenzi. Každé z dílčích hledání přitom vrací nejen nejlepší



Obrázek 3.9: *Kombinatorická reprezentace problému klasifikace paketů.*

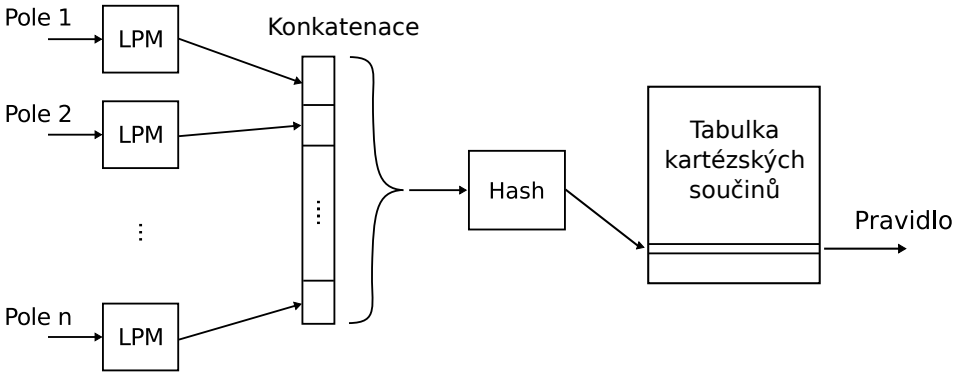
nalezený prefix, ale množinu prefixů tak, aby bylo možné při kombinaci s dalšími dílčími výsledky nalézt takové pravidlo, které je nejlepší nejen v jediné dimenzi, ale vyhovuje i ostatním dimenzím. K tomu je třeba vytvořit kartézský součin těchto množin a následně určit, které prvky kartézského součinu odpovídají některému pravidlu. Výsledkem algoritmu je pak takové pravidlo, které má nejvyšší prioritou.

Z výše uvedeného principu vycházejí tzv. dekompoziční algoritmy. Metody založené na dekompozici využívají myšlenky rozdělení problému na jednodušší podčásti, které mohou být zpracovány nezávisle. Problém klasifikace se řeší ve dvou základních krocích a odpovídá kombinatorickému pohledu na klasifikaci. V prvním kroku je provedeno několik vyhledání v jednotlivých dimenzích. Z nich je pak kombinováním vytvořen celkový výsledek klasifikace, který již jednoznačně identifikuje nalezené pravidlo. Dílčí vyhledávání jsou přitom nezávislá a vyhledání v každé dimenzi je možné provádět paralelně. Pro každou dimenzi lze také použít jinou dílčí metodu, která nejlépe odpovídá typu hodnot v dané dimenze. Z tohoto pohledu lze i dříve představený algoritmus BitVector řadit do skupiny dekompozičních algoritmů.

3.5.1 Kartézský součin

První z uvedených algoritmů vychází ze zjištění, že počet unikátních prefixů v rámci každé dimenze je výrazně nižší než celkový počet klasifikačních pravidel. Jak název algoritmu napovídá, ve fázi předzpracování se vytváří kartézský součin množin unikátních prefixů z každé dimenze. Dále je konstruována hashovací tabulka, která pro každý prvek kartézského součinu obsahuje předpočítané pravidlo s nejlepší shodou. Pro hledání v jednotlivých dimenzích jsou také vytvořeny vhodné datové struktury, například jednodimenzionální trie pro určení nejdelšího shodného prefixu nad jednotlivými položkami z hlaviček paketu.

Základní struktura algoritmu je zobrazena na obrázku 3.10. Samotná klasifikace algoritmem kartézského součinu se provádí následovně. Pro extrahované hodnoty z hlaviček paketů (pole 1 až n) je v každé dimenzi vyhledán nejdelší prefix (blok LPM). Výsledky jsou spojeny do jednoho datového slova (konkatenace), které slouží jako klíč a je vstupem hashovací funkce (blok hash). Výsledek funkce je potom indexem do přichystané tabulky (na položku s odpovídajícím pravidlem).



Obrázek 3.10: *Základní struktura algoritmu kartézského součinu (vyhledání LPM v jednotlivých dimenzích, konkatenace výsledků, výpočet hash a přístup do tabulky).*

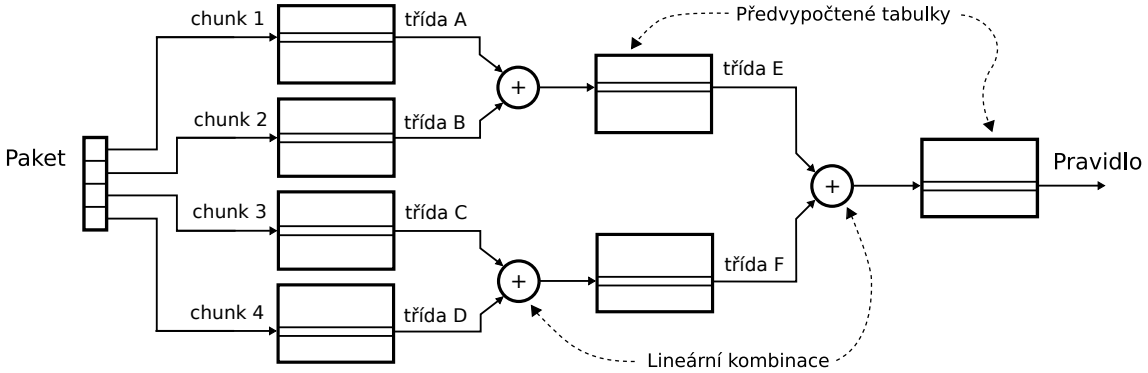
Časová složitost algoritmu závisí pouze na rychlosti vyhledávání v jednotlivých dimenzích. Dohledání nejlepšího pravidla lze při vhodně zvolené hashovací funkci pak provést jediným přístupem do paměti. Pokud zanedbáme výpočet hashovací funkce lze výsledné pravidlo získat v konstantním čase. Vyhledávání v jednotlivých dimenzích lze přitom provádět paralelně. Už z povahy kartézského součinu je však největším problémem algoritmu jeho paměťová složitost. Velikost konstruované tabulky roste exponenciálně. Její prostorová složitost je pro nejhorší případ $\mathcal{O}(n^k)$, kde n je počet unikátních prefixů a k počet dimenzí.

3.5.2 RFC

Další z algoritmů Recursive Flow Classification (RFC) [8] vychází z naivního přístupu přímého mapování paketů do tabulky pravidel, kde klasifikovaný paket slouží jako index do této tabulky. Hlavní odlišností od naivního přístupu je však to, že RFC využívá síť menších hierarchicky organizovaných tabulek a postupuje rekurzivně. Metoda je také jistou kombinací kartézského součinu a přístupu využívajícího bitový vektor a byla motivována zjištěním, že některé kombinace prefixů z jednotlivých dimenzích, tj. prvky kartézského součinu, odpovídají stejné podmnožině klasifikačních pravidel. Principem je výpočet částečných kartézských součinů, např. pro dvě dimenze, a jejich mapování na třídy ekvivalence.

Základní struktura algoritmu je schematicky znázorněna na obrázku 3.11. Pole z hlaviček paketů jsou rozdělena na datová slova (tzv. chunks) vhodné délky (například 16 bitů). Ta slouží k adresování jednotlivých tabulek, které jsou předvyplněny tak, aby se pakety odpovídající různým pravidlům mapovaly na různé hodnoty – třídy ekvivalence. Pro vytváření tříd ekvivalence se používají bitové vektory, tzn. prvky dílčího kartézského součinu, které mají stejný bitový vektor patřící vždy do jedné třídy ekvivalence. Dva různé prefixy tedy náleží do stejné třídy ekvivalence, pokud odpovídají stejnému klasifikačnímu pravidlu. Při samotném procesu klasifikace se již bitový vektor nepoužívá, slouží pouze pro konstrukci tříd ekvivalence. Výstupy z tabulek jsou dále spojovány lineární kombinací (na obrázku znázorněno symbolem $\oplus$) a slouží k adresování další úrovně tabulek. Celkový výsledek klasifikace se tedy nevytváří skládáním částečných kartézských součinů, ale kombinací odpovídajících tříd ekvivalence. Poslední tabulka v hierarchii nakonec provádí mapování třídy ekvivalence na identifikátor pravidla.

Při samotné klasifikaci dosahuje algoritmus velmi dobré propustnosti díky jednoduchosti prováděných operací, jakými jsou přímý přístup do tabulky a lineární kombinace (násobení konstantou a součet). Rekurzivní klasifikace toků minimalizuje exponenciální nárůst počtu



Obrázek 3.11: *Základní struktura algoritmu RFC (rozdělení hlaviček paketů na čtyři datová slova, adresace předpočítaných tabulek, lineární kombinace výsledků).*

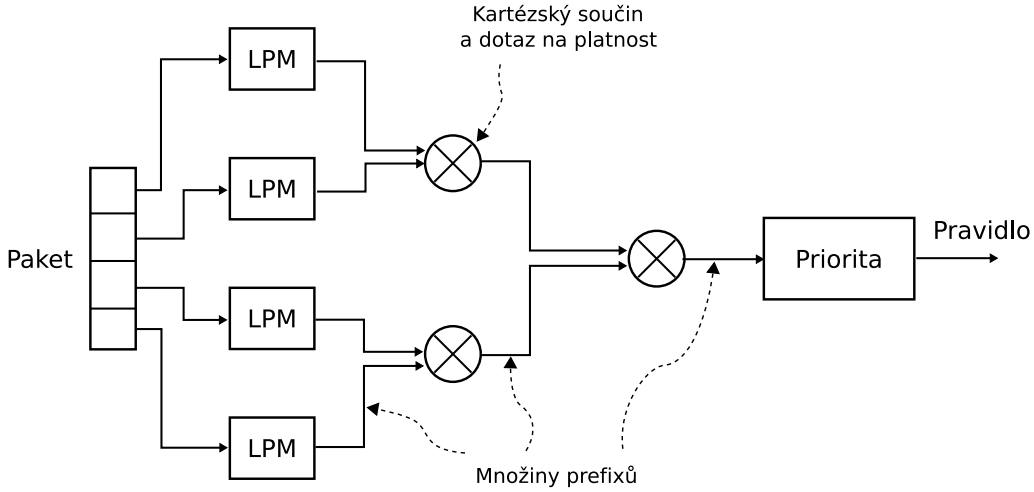
prvků kartézského součinu prohledávaných dimenzí a snižuje paměťové nároky díky klesajícímu počtu bitů, kterými se adresuje. I tak je však největším problémem algoritmu jeho paměťová náročnost.

3.5.3 DCFL

Také algoritmus Distributed Crossproducting of Field Labels [22] vychází ze základního přístupu kartézského součinu. Jeho základní strukturu ukazuje obrázek 3.12. Prvním krokem algoritmu je nezávislé vyhledávání prefixů, které odpovídají klasifikovanému paketu v každé dimenzi (blok LPM). Každé z dílčích hledání přitom vrací nejen nejdelší (nejspecifičtější) nalezený prefix, ale množinu všech, tedy i obecnějších prefixů. Druhým krokem je kombinování výsledků a ověřování, zda prvek kartézského součinu odpovídá některému pravidlu (na obrázku označeno symbolem $\otimes$). Vyhodnocování kartézského součinu přitom probíhá distribuovaně v rámci stromové struktury, tzv. agregační sítě. Výpočet se provádí postupně, vždy pro dvojici množin. V rámci každého stupně výpočtu se vždy ověřuje, zda se daná kombinace prefixů vyskytuje v některém klasifikačním pravidle. Některé prvky částečného kartézského součinu tak mohou být co nejdříve vyloučeny z dalšího zpracování, neboť daná kombinace prefixů neodpovídá žádnému klasifikačnímu pravidlu. To může mít vzhledem k charakteru kartézského součinu zásadní vliv na snížení doby výpočtu. Výstupem posledního stupně je množina odpovídajících pravidel, ze které je třeba vybrat to s nejvyšší prioritou.

Pro dotazování na existenci dané kombinace prefixů se používají tzv. Bloomovy filtry. Bloomův filtr [3] je pravděpodobnostní datová struktura, která je určena pro testování příslušnosti prvků do množiny. Samotné prvky množiny nejsou ve struktuře uloženy, takže její implementace může být z hlediska paměťové složitosti velmi efektivní. Jelikož jde o pravděpodobnostní strukturu, může docházet k chybám typu false positive, tzn. prvek je označen, že do množiny patří, ale ve skutečnosti není jejím prvkem. Opačná situace však nemůže nikdy nastat.

Ačkoliv lze prohledání jednotlivých dimenzí i vyhodnocování částečných kartézských součinů provádět paralelně, největší nevýhodou algoritmu je právě nutnost výpočtu kartézského součinu a potřeba pro každý prvek součinu ověřovat jeho přítomnost v dané množině. Z tohoto přístupu vyplývá také nepříjemný problém nedeterministické časové složitosti, která může pro určité případy klasifikace výrazně narůst. Nevýhodou je také možnost chyby při použití Bloomových filtrů, kterou je potřeba zohlednit v dalších krocích výpočtu.



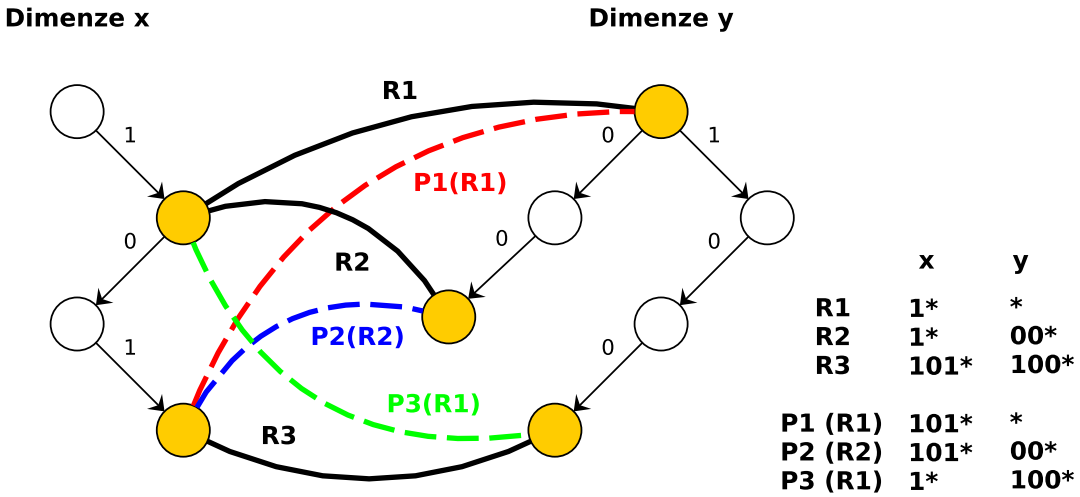
Obrázek 3.12: *Základní struktura algoritmu DCFL (klasifikace pro 4 dimenze, kombinování množin prefixů z jednotlivých dimenzí, výběr pravidla s nejvyšší prioritou); [2].*

3.5.4 MSCA

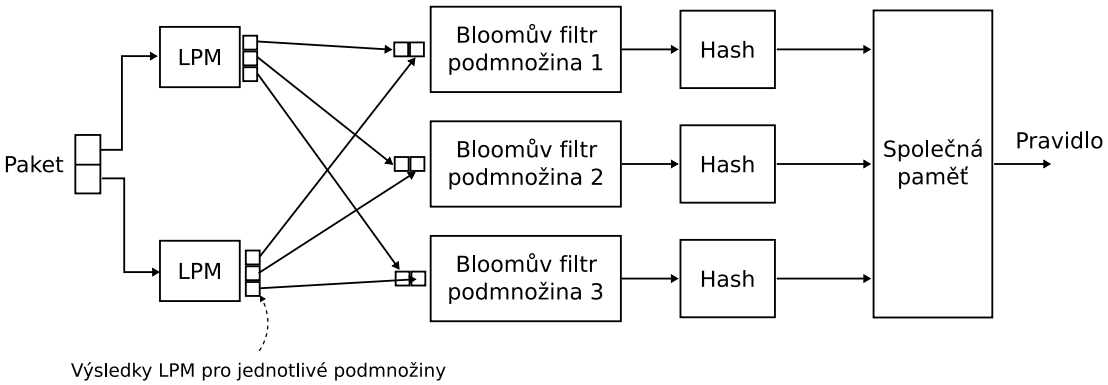
Další z modifikací základního algoritmu kartézského součinu je Multi Subset Crossproduct Algorithm (MSCA) [5], který zavádí nový pojem tzv. pseudopravidlo. Kartézský součin je tvořen kombinací unikátních hodnot ze všech dimenzí. Ačkoliv všechny prvky kartézského součiny nutně netvoří nějaké klasifikační pravidlo, typicky nějakému pravidlu odpovídají. Právě takové záznamy jsou označovány jako pseudopravidla. Pseudopravidlo je takové pravidlo, které je potřeba do množiny klasifikačních pravidel přidat, aby byla každá regulérní kombinace výsledků operace LPM v jednotlivých dimenzích pokryta. Ilustraci vzniku pseudopravidel ukazuje obrázek 3.13 na příkladu sady tří klasifikačních pravidel R_1, R_2 a R_3 z obrázku 3.9. Původní pravidla R_1, R_2 a R_3 jsou zobrazena plnou černou čárou. Přidaná pseudopravidla P_1, P_2 a P_3 jsou zobrazena barevně, čárkovaně. V závorce je přitom vždy uvedeno příslušné pravidlo, kterému odpovídají.

Algoritmus MSCA přichází s přístupem rozdělení množiny pravidel do několika podmnožin, který právě počet vznikajících pseudopravidel minimalizuje. Při rozdělení je na první pohled nutné operaci LPM pro každou podmnožinu provádět odděleně. Výhodnější alternativou je však prosté uložení výsledku LPM zvlášť pro každou z podmnožin. Základní strukturu algoritmu ilustruje obrázek 3.14. Na obrázku je znázorněna klasifikace pro dvě dimenze využívající navíc rozdělení sady pravidel do tří podmnožin. Prvním krokem klasifikace je provedení operace LPM zvlášť pro každou z dvojice dimenzí. Jelikož není zřejmé, ve které podmnožině se nachází platné pravidlo, je nutné pro každou podmnožinu provádět výpočet hashovací funkce (blok hash) a přístup do společné paměti. Počet potřebných přístupů do paměti lze zredukovat předřazením Bloomova filtru pro každou podmnožinu.

Algoritmus MSCA je velmi výhodný z pohledu rychlosti i paměťové náročnosti. Nevýhodou je však možnost chyby typu false positive při použití Bloomových filtrů, kterou je třeba řešit kontrolou pravidla v hlavní paměti. To může v nepříznivé situaci ovlivnit propustnost systému z důvodů zvýšeného počtu přístupů do paměti. Kromě rozdělení množiny pravidel na podmnožiny navrhuje algoritmus také alternativní přístup pro snížení celkového počtu pseudopravidel. Pravidla, která způsobují největší nárůst pseudopravidel (označovaná jako spoilers) je totiž možné umístit do malé paralelně prohledávané TCAM.



Obrázek 3.13: *Vznik pseudopravidel na příkladu z obrázku 3.9.*

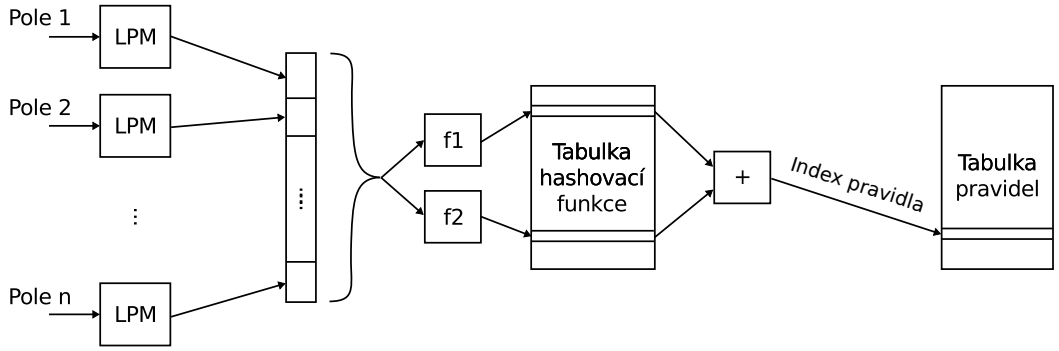


Obrázek 3.14: *Základní struktura algoritmu MSCA (ukázka klasifikace pro dvě dimenze a rozdělení množiny pravidel do tří podmnožin); [2].*

3.5.5 PHCA

Poslední z uvedených přístupů Perfect Hashing Crossproduct Algorithm (PHCA) [15] přichází s myšlenkou odstranění potřeby ukládat pseudopravidla. Principem algoritmu je konstrukce perfektní hashovací funkce, která umožní přímé mapování výsledků dílčích operací LPM na odpovídající pravidlo. Hashovací funkce je záměrně vytvořena tak, aby se všechna pseudopravidla mapovala přesně na pravidlo, kterému odpovídají, a nebylo třeba je explicitně ukládat.

Struktura algoritmu je zobrazena na obrázku 3.15. Samotný proces klasifikace potom probíhá ve třech krocích. V rámci prvního kroku je v každé dimenzi a pro každé pole (1 až n) vyhledán nejdelší shodný prefix (blok LPM). Výsledky jsou v druhém kroku spojeny do jednoho datového slova, které slouží jako vstup do dvou běžných hashovací funkcí (na obrázku označeny jako f_1 a f_2). Jejich výstup slouží k indexaci předpočítané tabulky definující hodnotu vytvořené perfektní hashovací funkce, která určuje index odpovídajícího pravidla v tabulce pravidel. Posledním krokem algoritmu je ověření správnosti určeného



Obrázek 3.15: *Základní struktura algoritmu PHCA (poslední krok algoritmu – ověření správnosti nalezeného pravidla není zobrazen); [2].*

pravidla přímým porovnáním s klasifikovaným paketem. Poslední krok je potřeba provádět z důvodu možné existence paketů, které neodpovídají žádnému pravidlu. Tato situace totiž není ošetřena a hashovací funkce takovému paketu nějakého pravidlo přesto přiřadí.

Časovou složitost všech kroků algoritmu PHCA lze považovat za konstantní:

Vyhledání LPM v jednotlivých dimenzích. Časová složitost vyhledání LPM je lineární vzhledem k počtu bitů datového slova, avšak počet bitů každé dimenze je pevný a předem známý, proto lze LPM považovat za operaci s konstantní složitostí.

Výpočet hashovací funkce. Vyčíslení perfektní hashovací funkce spočívá ve výpočtu dvou běžných hashovacích funkcí, dvou přístupech do paměti a provedení aritmetického součtu. Všechny tyto dílčí kroky mají konstantní časovou složitost.

Ověření správnosti nalezeného pravidla. Poslední krok spočívá v jediném přístupu do paměti a porovnání položek pevné velikosti. Má také konstantní časovou složitost.

Algoritmus PHCA je proto velmi výhodný z pohledu rychlosti. Operace LPM v jednotlivých dimenzích lze provádět paralelně a pro klasifikaci každého paketu stačí konstantní počet přístupů do paměti. Značnou nevýhodou algoritmu je však jeho paměťová náročnost.

Kapitola 4

Analýza klasifikačních algoritmů

Následující část práce popisuje výsledky detailnější analýzy vybraných klasifikačních algoritmů představených v předcházející kapitole. Při provádění analýzy byla použita knihovna Netbench [17]. Netbench je specializovaná knihovna v jazyce Python, která slouží pro snadnou implementaci algoritmů z oblasti zpracování paketů a umožňuje provádění experimentů, vyhodnocování a srovnávání jejich vlastností. V dnešní době poskytuje knihovna programové rozhraní nejen pro implementaci klasifikace paketů. Obsahuje také podporu pro hledání nejdelšího shodného prefixu (longest prefix match) a vyhledávání řetězců (pattern matching). Součástí knihovny je i řada referenčních implementací známých algoritmů a testovacích datových sad.

Pro vyhodnocení algoritmů BitVector, HiCuts, MSCA, DCFL a PHCA jsem s výhodou použil právě knihovnu Netbench. Pro algoritmus HyperSplit pak byla použita vlastní implementace vycházející z referenční implementace, odkazované v již uvedené původní publikaci [26].

Analýza probíhala metodou experimentů s jednotlivými algoritmy a konkrétními sadami klasifikačních pravidel, které pocházejí nebo byly odvozeny z reálné množiny pravidel používaných při filtraci paketů. Získané výstupy jednotlivých algoritmů jsou následně vyhodnoceny. Hlavním ze sledovaných parametrů byla paměťová náročnost každého z algoritmů, neboť ta je jedním z nejdůležitějších parametrů. Na jedné straně je kapacita rychlých pamětí dostupných přímo na čipu omezená, na druhé straně použití dostatečně velké externí paměti zase negativně ovlivňuje výslednou propustnost celého řešení. Je to z důvodu vyšší latence nebo také nižší propustnosti externích pamětí. Čipy FPGA sice obsahují určité množství blokových pamětí, ty se ale zdaleka svou kapacitou nevyrovnají paměti externí.

U některých algoritmů bylo možné dále sledovat i jiné parametry. Jednalo se například o množství ukládaných pseudopravidel nebo výšku zkonstruovaného rozhodovacího stromu, která přímo ovlivňuje počet přístupů do paměti a výslednou časovou složitost algoritmu. Každý z přístupů je ale specifický a další získané výstupy jsou již hůře srovnatelné.

4.1 Použité sady klasifikačních pravidel

Měření probíhalo na datových sadách pocházejících ze dvou zdrojů. Prvním zdrojem jsou volně dostupné sady pravidel [21] odkazované z řady publikací věnujících se klasifikačním algoritmům např. [26]. Druhým zdrojem jsou pak testovací datové sady z knihovny Netbench. Pro testování vlastností vybraných algoritmů bylo použito celkem 29 datových sad. Ty se liší jednak počtem pravidel, dále pak počtem unikátních hodnot v jednotlivých dimenzích.

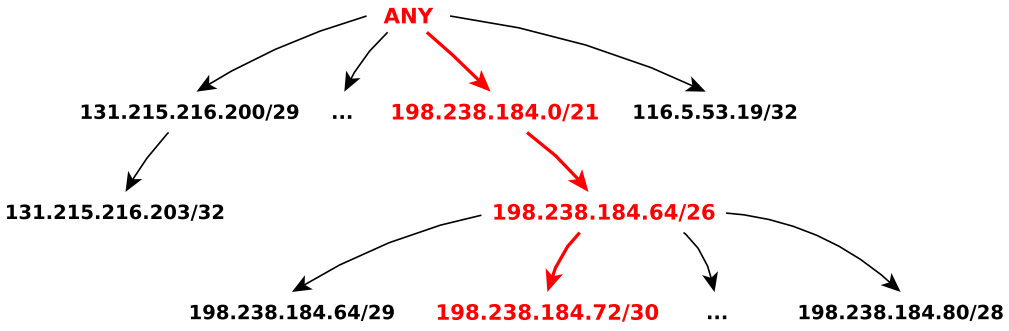
Tabulka 4.1: *Základní vlastnosti použitých datových sad filtračních pravidel.*

Datová sada	Počet pravidel	Src IP	Dst IP	Src port	Dst port	Proto
acl1	753 / 1 554	72 (4)	161 (4)	1 (1)	283 (8)	4 (2)
acl1-100	99 / 130	34 (4)	62 (5)	1 (1)	63 (3)	4 (2)
acl1-1K	917 / 1 215	103 (4)	297 (4)	1 (1)	164 (5)	4 (2)
acl1-5K	4 415 / 6 025	805 (4)	640 (6)	1 (1)	182 (5)	4 (2)
acl1-10K	9 603 / 12 904	4 784 (4)	733 (6)	1 (1)	182 (5)	4 (2)
fw1	270 / 870	33 (3)	64 (4)	22 (3)	53 (3)	5 (2)
fw1-100	93 / 303	12 (4)	25 (2)	21 (3)	36 (3)	5 (2)
fw1-1K	792 / 3 307	124 (4)	175 (4)	23 (3)	52 (3)	5 (2)
fw1-5K	4 653 / 15 778	1 746 (4)	2 714 (4)	23 (3)	53 (3)	5 (2)
fw1-10K	9 311 / 32 136	3 638 (3)	6 951 (2)	23 (3)	53 (3)	5 (2)
ipc1	1 550 / 1 773	136 (4)	117 (4)	38 (3)	51 (3)	7 (2)
ipc1-100	100 / 146	63 (4)	63 (5)	20 (3)	19 (3)	4 (2)
ipc1-1K	938 / 1 223	336 (4)	436 (5)	35 (4)	58 (4)	6 (2)
ipc1-5K	4 460 / 5 916	585 (4)	1 251 (6)	41 (4)	57 (4)	7 (2)
ipc1-10K	9 037 / 12 127	1 515 (4)	2 726 (6)	41 (4)	67 (4)	7 (2)
nb1-m05-m05	852 / 2 261	121 (4)	143 (5)	23 (3)	51 (3)	2 (2)
nb1-m05-05	827 / 3 195	133 (4)	146 (4)	23 (3)	53 (3)	2 (2)
nb1-05-m05	882 / 2 326	156 (4)	171 (5)	23 (3)	53 (3)	2 (2)
nb1-05-05	822 / 3 552	138 (4)	123 (4)	23 (3)	53 (3)	2 (2)
nb2-m05-m05	992 / 1 627	826 (3)	425 (3)	14 (2)	1 (1)	2 (2)
nb2-m05-05	984 / 2 162	802 (3)	412 (3)	14 (2)	1 (1)	2 (2)
nb2-05-m05-100	100 / 1 827	75 (3)	37 (3)	13 (2)	1 (1)	2 (2)
nb2-05-m05-250	240 / 159	55 (3)	53 (3)	14 (2)	1 (1)	2 (2)
nb2-05-m05-500	455 / 374	65 (3)	57 (3)	14 (2)	1 (1)	2 (2)
nb2-05-m05	962 / 649	746 (3)	307 (3)	14 (2)	1 (1)	2 (2)
nb2-05-05	956 / 2 152	735 (3)	274 (3)	14 (2)	1 (1)	2 (2)
nb3-05-05-500	472 / 1 731	103 (4)	52 (3)	19 (3)	49 (3)	2 (2)
nb4-05-05-500	482 / 3 791	54 (3)	82 (4)	48 (3)	62 (3)	2 (2)
nb5-05-05-500	481 / 1 397	105 (5)	84 (4)	21 (3)	47 (3)	2 (2)

Charakteristiku jednotlivých datových sad zobrazuje tabulka 4.1. Tabulka je rozdělena na dvě části, které odpovídají použitým zdrojům pravidel.

Datové sady označené jako `acl1`, `fw1` a `ipc1` byly původně extrahovány z reálných množin filtračních pravidel a odpovídají po řadě zdrojovým formátům *Access Control List (ACL)*, *Firewall (FW)* a *IP Chain (IPC)*. Pojmenování dalších datových sad `100`, `1K`, `5K` a `10K` poté značí počet obsažených pravidel – sto, jeden tisíc, pět tisíc a deset tisíc pravidel. Tyto odvozené sady již byly vytvořeny pomocí nástroje ClassBench [23]. Ten slouží právě ke generování syntetických datových sad určených pro testování algoritmů z oblasti klasifikace síťového provozu.

Druhá skupina datových sad označená jako `nb1` až `nb5` odpovídá testovacím sadám z knihovny Netbench. Ty byly původně také vygenerovány a jsou tak rovněž syntetické. Jejich detailnější označení odpovídá parametrům *address scope* a *application scope* použitým při generování nástrojem ClassBench.



Obrázek 4.1: *Počet úrovní stromu prefixů zdrojových adres pro sadu* `fw1-100`*.*

Všechny použité datové sady obsahují klasifikační pravidla s pěticí dimenzí tvořenou prefixy zdrojové a cílové IPv4 adresy, typem transportního protokolu a rozsahy zdrojového a cílového portu.

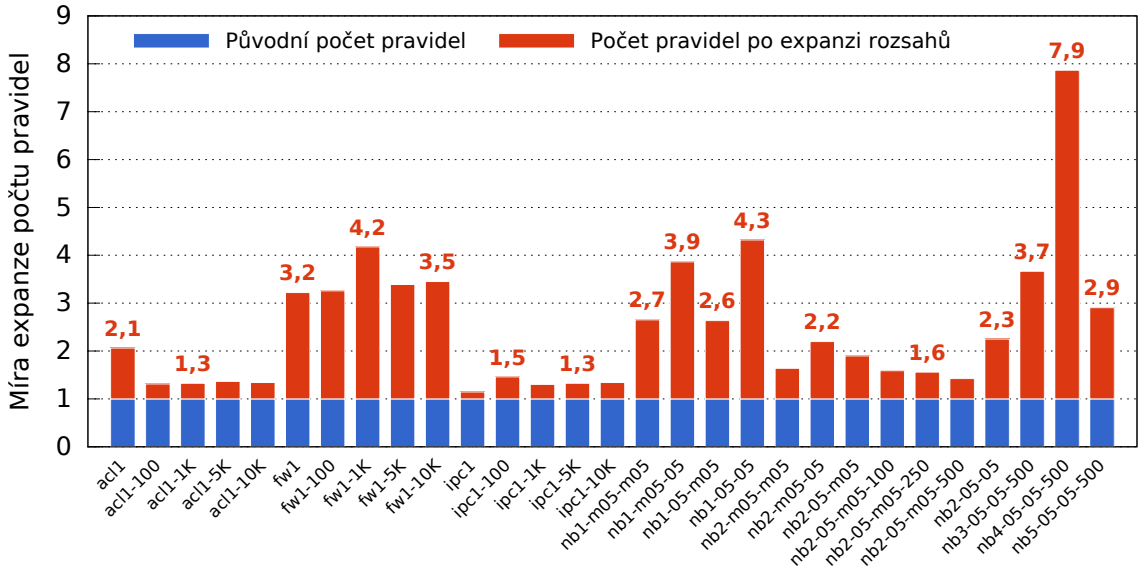
První sloupec tabulky 4.1 obsahuje počet pravidel dané datové sady. Hodnota za lomítkem vyjadřuje expanzi počtu pravidel po převedení rozsahů portů na prefixy. V následujících sloupcích je uveden počet všech unikátních hodnot prefixů v jednotlivých dimenzích. Hodnota v závorce pak souvisí s počtem úrovní prefixového stromu a vyjadřuje maximální počet prefixů dimenze, které mohou odpovídat paketu při klasifikaci. Význam hodnoty blíže ilustruje obrázek 4.1 s příkladem stromu prefixů zdrojových IP adres z datové sady `fw1-100`.

Při klasifikaci paketu se zdrojovou IP adresou například *198.238.184.73* tomuto paketu v dimenzi zdrojových adres odpovídají následující čtyři prefixy (na obrázku znázorněny červeně): *198.238.184.72/30*, *198.238.184.64/26*, *198.238.184.0/21* a univerzální hodnota *ANY*. Maximální možný počet takových prefixů pro libovolný paket udává právě hodnota uvedená v závorce.

Velmi důležitou vlastností vyplývající z tabulky 4.1 je, že celkový počet unikátních prefixů v jednotlivých dimenzích i maximální možný počet prefixů odpovídající klasifikovanému paketu je vždy výrazně menší než celkový počet pravidel dané sady. Maximální počet prefixů odpovídající klasifikovanému paketu je také výrazně nižší než celkový počet unikátních prefixů. A co je ještě důležitější, s rostoucím počtem pravidel datové sady se maximální počet prefixů prakticky vůbec nemění.

Další z charakteristických vlastností datových sad je velmi malá množina hodnot v poli protokolu. Většina pravidel obsahuje hodnoty *TCP*, *UDP* nebo univerzální hodnotu *ANY*. Zřídkakdy se používá typ *ICMP* nebo jiné hodnoty.

Pro zdrojový a cílový port je zase charakteristické intervalové porovnání. Velké množství pravidel z datových sad přitom obsahuje buď univerzální hodnotu *ANY* nebo porovnání na privilegované (tzv. well known) porty s rozsahy $\langle 0, 1023\rangle$ nebo $\langle 1024, 65535\rangle$. Potřeba převádět rozsahy na prefixové vyjádření v tomto případě nemusí být úplně nejvhodnější. Například zmíněný interval $\langle 1024, 65535\rangle$ je totiž bezpodmínečně nutné vyjádřit pomocí celkem šesti prefixových intervalů $\langle 1024, 2047\rangle$, $\langle 2048, 4095\rangle$, $\langle 4096, 8191\rangle$, $\langle 8192, 16383\rangle$, $\langle 16384, 32767\rangle$ a $\langle 32768, 65535\rangle$. Nárůst počtu pravidel při převodech rozsahů na prefixy pro použité datové sady ukazuje graf na obrázku 4.2. Na ose x jsou uvedeny jednotlivé datové sady. Na ose y je potom znázorněna míra expanze množství pravidel vzhledem k jejich původnímu počtu. Pro testované datové sady dochází k průměrnému nárůstu 2,5 krát, v nejhorším případě téměř 8 krát.



Obrázek 4.2: *Míra expanze počtu pravidel při převodu rozsahů na prefixy.*

4.2 Paměťová náročnost vybraných algoritmů

Určená celková spotřeba paměti pro jednotlivé testovací sady pravidel a vybrané algoritmy je zobrazena v tabulce 4.2. Veškeré hodnoty jsou uvedeny v kilobajtech. Protože se datové struktury používané při výpočtu u některých metod nevešly do dostupné operační paměti, není tabulka kompletní. V případech, kdy nebylo možné výsledek určit a výpočet musel být předčasně ukončen, je v tabulce uvedena pomlčka. Tento problém se však týkal pouze velkých datových sad, tj. sad s vyšším počtem pravidel. Jednou z hlavních příčin je použití skriptovacího jazyku Python, druhým z důvodů je pak typicky samotný algoritmus, který má při konstrukci datových struktur příliš velké paměťové nároky.

Nejlepších výsledků při srovnání dosahuje algoritmus DCFL, v řadě případů i řádově lepších. Z hlediska paměťové náročnosti se tak jeví jako nejvýhodnější. Dekompoziční algoritmus DCFL totiž nepracuje jen s nejdelším prefixem, ale celou množinou všech, i obecnějších prefixů. Vlivem následného provádění kartézského součinu množin prefixů má tak nejmenší paměťové nároky. Nemusí totiž řešit problém ukládání pseudopravidel. To však přináší výrazně velkou časovou náročnost vlivem nutnosti postupného vyčíslování kartézského součinu.

Algoritmus BitVector rozděluje každou dimenzi na množství intervalů a každému z nich přiřazuje bitový vektor reprezentující odpovídající klasifikační pravidla. Kombinování výsledků z jednotlivých dimenzí se provádí velmi efektivně pomocí operace bitového součinu. Algoritmus je také poměrně efektivní pro malé množiny pravidel. S narůstajícím počtem pravidel ale dochází k výraznému nárůstu paměťové složitosti. Dalším z problémů algoritmu je pak jeho časová složitost a náročnost na hardwarové zdroje, viz navazující kapitola 4.3.

Další dva srovnávané přístupy MSCA a PHCA dosahují z pohledu paměti obdobných výsledků, neboť vychází z podobných principů. Oba jsou založeny na myšlence dekompozice. Algoritmus MSCA ukládá do paměti jak původní pravidla, tak vznikající pseudopravidla. Metoda PHCA pseudopravidla do paměti neukládá, ale využívá konstrukce perfektní hashovací funkce. Pro snížení celkového počtu vznikajících pseudopravidel počítají dále oba přístupy s přesunem 8 pravidel, tzv. spoilers (pravidel, které způsobují největší nárůst počtu

Tabulka 4.2: *Paměťová náročnost vybraných algoritmů (hodnoty v kB).*

Datová sada	HyperSplit	HiCuts	MSCA	PHCA	BitVector	DCFL
acl1	14,8	84,3	8 189,9	4 730,0	44,8	10,2
acl1-100	3,1	6,1	57,6	23,7	4,6	2,2
acl1-1K	29,7	133,8	8 917,6	3 471,5	103,7	10,7
acl1-5K	149,8	–	–	–	1 361,5	–
acl1-10K	315,3	–	–	–	11 126,1	–
fw1	9,4	2 398,9	480,7	2 659,8	9,4	3,3
fw1-100	3,1	434,8	65,2	307,6	2,3	3,9
fw1-1K	83,2	64 794,7	85 014,4	175 082,9	66,6	15,3
fw1-5K	3 183,9	–	–	–	5 060,3	–
fw1-10K	12 750,5	–	–	–	24 352,5	–
ipc1	44,9	917,0	12 946,9	–	93,6	16,9
ipc1-100	3,2	5,7	576,8	336,5	5,6	3,3
ipc1-1K	32,7	195,6	276 637,4	–	182,0	12,1
ipc1-5K	200,1	–	–	–	1 752,7	–
ipc1-10K	447,9	–	–	–	7 938,2	–
nb1-m05-m05	116,7	7 663,2	53 262,4	37 330,0	52,1	11,8
nb1-m05-05	119,7	9 516,8	82 272,5	48 813,3	51,8	13,9
nb1-05-m05	322,2	16 312,6	51 966,0	38 876,8	69,0	12,5
nb1-05-05	265,7	16 488,2	58 955,6	56 000,9	49,0	15,2
nb2-m05-m05	89,9	353,3	8 203,5	6 623,1	313,0	10,7
nb2-m05-05	92,2	507,3	8 103,7	5 586,9	301,2	11,4
nb2-05-m05-100	3,4	8,6	23,9	11,0	3,7	1,9
nb2-05-m05-250	8,1	48,7	81,8	36,7	5,0	1,9
nb2-05-m05-500	14,4	80,0	147,6	62,1	7,9	3,0
nb2-05-m05	113,6	540,8	5 246,3	4 449,1	253,1	10,4
nb2-05-05	124,7	801,3	1 097,2	4 637,7	239,7	10,9
nb3-05-05-500	26,1	1 220,0	1 303,8	8 833,5	16,4	12,7
nb4-05-05-500	43,0	2 197,7	10 828,2	28 212,0	16,7	28,7
nb5-05-05-500	43,3	1 237,6	19 495,2	20 215,0	18,2	10,2

pseudopravidel), do malé paralelně prohledávané paměti TCAM. MSCA využívá navíc přístup rozdělení klasifikačních pravidel do podmnožin. Výsledné využití paměti je ale nakonec u obou přístupů značně ovlivněno počtem vznikajících pseudopravidel, i když s nimi každá z metod nakládá jinak. Ve snaze dosáhnout konstantní časové složitosti mají oba algoritmy velmi velké paměťové nároky a směřují k použití externích pamětí s velkou kapacitou.

S příchodem nových technologií FPGA UltraSCALE a UltraSCALE+ [25] množství paměti dostupné na čipu narůstá. Při volbě vhodné klasifikační metody je tak možné umístit všechna pravidla jen na čip, bez nutnosti využití externí paměti. Z tohoto pohledu je tedy zajímavé se zaměřit na redukci paměťové složitosti. Tu umožňují například zkoumané metody HiCuts a HyperSplit založené na principu konstrukce rozhodovacích stromů. Při srovnání s dekompozičními metodami MSCA a PHCA vykazují oba přístupy značnou úsporu paměti. Algoritmus HyperSplit pak umožňuje ještě výrazně vyšší redukci paměťové složitosti díky technice rozsahového porovnávání, která přispívá také lepší vyváženosti kon-

struovaného rozhodovacího stromu. Nezávislost těchto metod na použití externí paměti, přímočaré odvození časové složitosti vycházející z výšky konstruovaného stromu a možnost mapování algoritmů do hardwarové architektury se zřetězenou strukturou z nich činí vhodné kandidáty pro filtraci paketů na 100 Gb sítích.

4.3 Časová náročnost vybraných algoritmů

Po paměťové náročnosti je další z velmi důležitých vlastností klasifikačních algoritmů dosahovaná propustnost. Propustnost je typicky měřena jako počet klasifikovaných paketů za jednu sekundu. Dosažená propustnost je však silně závislá na konkrétní implementaci daného algoritmu. Závisí na použité konkrétní technologii, množství zabraných zdrojů a dosažené pracovní frekvenci. Rozhodujícím faktorem ovlivňujícím výslednou propustnost je bezesporu také typ použité paměti (SRAM, DRAM), zda je využita paměť dostupná na čipu či paměť externí, jestli se potřebné datové struktury vejdou do paměti cache či nikoliv.

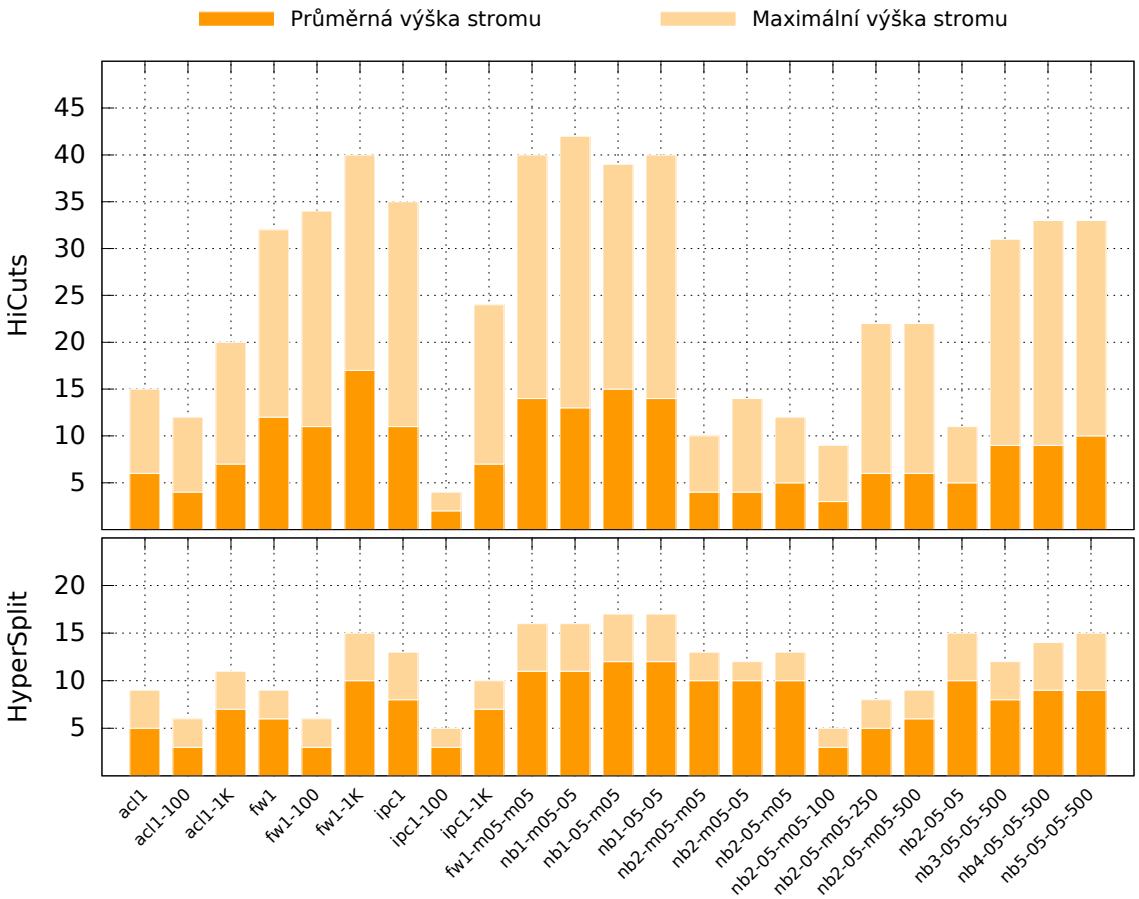
O propustnosti v této podobě má tak smysl hovořit až při zkoumání cílového řešení – konkrétní výsledné architektury pro filtraci paketů. Z praktického hlediska je tak při srovnávání rychlosti klasifikačních algoritmů vhodné sledovat spíše počet výpočetních kroků, případně počet přístupů do paměti potřebný pro provedení klasifikace jednoho paketu.

Z pohledu časové složitosti lze nejlepší výsledky očekávat u algoritmů založených na dekompozici a vycházejících z naivního přístupu konstrukce kartézského součinu (MSCA, PHCA). Tyto algoritmy se totiž nejvíce blíží konstantní časové složitosti vyhledávání. Paměť potřebná pro uložení veškerých dat však pro tyto a další příbuzné algoritmy roste exponenciálně s počtem pravidel vlivem odpovídajícího nárůstu množství pseudopravidel. Algoritmy sice umožňují a jsou navrženy pro využití externí paměti, pro filtraci paketů na vysokorychlostních sítích (pro rychlosti 100 Gb/s a více) je ale použití externích pamětí nevhodné. Důvodem je vysoká latence a nízká propustnost externích pamětí. Z tohoto pohledu je tak mnohem výhodnější zaměřit se na efektivní využití rychlých pamětí dostupných přímo na čipu.

U dalších algoritmů, jejichž postup výpočtu nemusí být tak přímočarý, musí analýza časové složitosti nutně počítat s nejhorší situací ve všech fázích výpočtu. Nejhorší případ se přitom může významně odlišovat od typického.

Dekompoziční algoritmus DCFL dosahuje velmi dobrých výsledků v oblasti paměťové náročnosti. Pro filtraci paketů na vysokorychlostních sítích je však zcela nevhodný právě z důvodu nedeterministické časové složitosti. Pro určité případy klasifikace paketu totiž vyžaduje až 40 přístupů do paměti, což výrazně a neúměrně snižuje celkovou propustnost takového řešení. Bez výrazných modifikací není možné takový přístup pro zpracování a filtraci síťového provozu s rychlostí 100 Gb/s a více použít.

Algoritmus BitVector lze pro malé množiny pravidel a hlediska paměťových nároků považovat za poměrně efektivní. Značnou nevýhodou tohoto přístupu je však výsledná velikost bitového vektoru, která roste s počtem klasifikačních pravidel. S narůstajícím počtem pravidel dochází k výraznému nárůstu nejen paměťové složitosti, ale také počtu přístupů do paměti, které jsou nutné pro načtení celého vektoru. Dalším z problémů algoritmu je, že v případě jeho hardwarové realizace je počet pravidel shora striktně omezen maximální velikostí vektoru, která je hardwarovou realizací podporována. U architektury dimenzované na nejhorší případ to nutně povede na velkou spotřebu hardwarových zdrojů. Z důvodu velké náročnosti na hardwarové zdroje a špatné škálovatelnosti je algoritmus v této podobně pro filtraci paketů na 100 Gb sítích nevhodný.



Obrázek 4.3: *Srovnání výšky rozhodovacího stromu u přístupů HiCuts a HyperSplit.*

Časová složitost dalších analyzovaných algoritmů HiCuts a HyperSplit odpovídá výšce konstruovaného rozhodovacího stromu. Srovnání průměrné a nejhorší časové složitosti a rozdíly mezi průměrnou a maximální výškou rozhodovacího stromu pro testované datové sady a oba dva algoritmy ukazuje graf na obrázku 4.3.

V horní části obrázku jsou znázorněny výsledky algoritmu HiCuts, v dolní části pak výsledky pro algoritmus HyperSplit. Na první pohled je patrné, že HiCuts vykazuje celkově vyšší hodnoty maximální výšky stromu a časové složitosti, přičemž průměrná výška stromu je poměrně nízká. U algoritmu HyperSplit se přitom hodnoty průměrné a maximální výšky stromu tolik neliší. Získané výsledky tak ukazují na lepší vyváženost a efektivitu rozhodovacího stromu konstruovaného přístupem HyperSplit. Algoritmus HyperSplit totiž umožňuje přesnější dělení prostoru díky intervalovému porovnávání a ukládání hraniční hodnoty v uzlech stromu. Naopak HiCuts umožňuje dělení prostoru na více částí stejné velikosti, což přispívá ke snížení průměrné výšky stromu, avšak vede ke vzniku ojediněle velmi dlouhých větví, které mají negativní vliv na časovou složitost algoritmu v extrémních situacích.

Oba algoritmy lze výhodně rozprostřít do pipeline a díky zřetězenému zpracování je tak možné při implementaci dosáhnout velmi efektivní paralelizace a rychlosti zpracování. To vše je zcela ve shodě s požadavky na filtraci paketů u sítí o rychlostech 100 Gb/s a 400 Gb/s. Pro zpracování síťového provozu na těchto sítích je velmi vhodné použití tzv. hlubokých pipelines (deep pipelines). Jedná se tedy o nespornou výhodu obou algoritmů.

4.4 Shrnutí

Dekompoziční algoritmy MSCA, PHCA a další přístupy z nich odvozené sice vykazují konstantní časovou složitost při vyhledávání, ale z důvodu vysokých paměťových nároků vyžadují použití externích pamětí. S nárůstem kapacity síťových linek proto nevyhovují požadavkům na rychlost klasifikace kvůli velké přístupové době do externí paměti.

Vývoj dnešních technologií přináší nárůst paměti dostupné na čipu, což umožňuje na čip umístit i velké množiny pravidel. Z těchto důvodů je tak důležité se dále zabývat klasifikačními algoritmy, které umožňující redukci paměťové složitosti a které dokáží efektivně využít pouze interní paměť dostupnou na čipu.

Paměťově velmi efektivní algoritmus DCFL je však pro potřeby filtrace paketů na 100 Gb sítích z důvodu nedeterministické časové složitosti nevhodný. Přístup BitVector zase vede na vysokou náročnost na hardwarové zdroje a špatnou škálovatelnost pro velké sady pravidel. Z těchto důvodů je výhodné se dále zaměřit na algoritmy HyperSplit a HiCuts založené na rozhodovacích stromech, které využívají pouze paměť dostupnou na čipu a které umožňují velmi efektivní řešení klasifikace díky možnosti zřetězeného zpracování a mapování do pipeline. Pro zpracování síťového provozu na sítích o rychlostech 100 Gb/s nebo 400 Gb/s je totiž velmi vhodné použít tzv. hluboké pipeline (deep pipelines).

S ohledem na charakteristické vlastnosti sad filtračních pravidel, kterými jsou nízký počet unikátních hodnot v jednotlivých dimenzích, nízký počet prefixů odpovídající klasifikovanému paketu a intervalové porovnání typické pro zdrojový a cílový port, dává smysl se i nadále zabývat dalšími optimalizacemi a rozvojem těchto přístupů, neboť stále existuje prostor pro dosažení ještě vyšší efektivity a propustnosti při zpracování a filtraci síťového provozu.

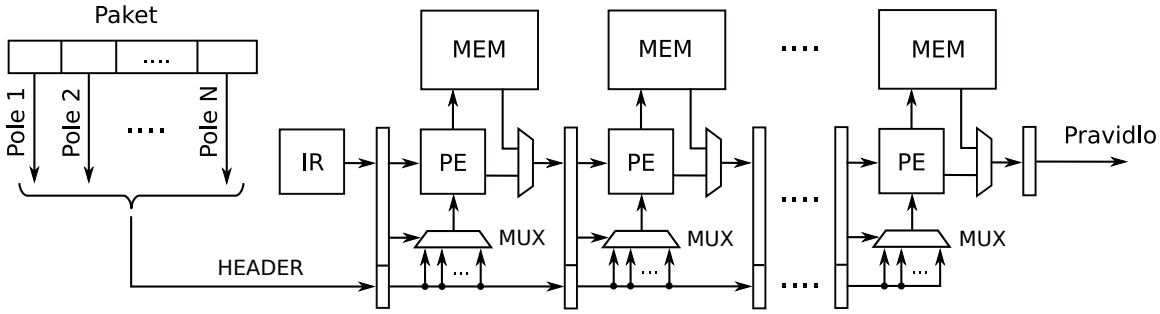
Kapitola 5

Navržená architektura

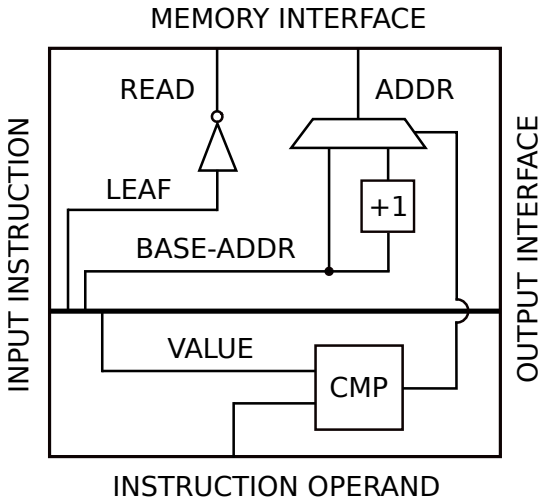
Hlavním požadavkem při návrhu hardwarové architektury pro filtraci paketů ve 100 Gb sítích je dosažení plné propustnosti linky (tzv. wire speed) a zajištění maximální redukce paměťové složitosti, která umožní efektivně využít pouze interní paměť dostupnou na čipu. Toho lze dosáhnout využitím algoritmů založených na rozhodovacích stromech, které jednak vykazují vynikající poměr mezi časovou a paměťovou složitostí a zároveň je možné je velmi dobře mapovat do tzv. hlubokých pipeline (deep pipelines), které jsou velmi výhodné právě pro zpracování síťového provozu ve vysokorychlostních sítích.

Na základě těchto požadavků jsem navrhl obecný způsob klasifikace paketů pro 100 Gb sítě. Přístup je založen na řetězené lince, která slouží k provádění programu složeného ze sekvence jednoduchých instrukcí. Program vlastně představuje jistou reprezentaci rozhodovacího stromu, jehož jednotlivé instrukce odpovídají původním uzlům stromu. Navržená architektura vychází z obecného principu mapování rozhodovacího stromu do pipeline, kdy je sekvenční průchod rozhodovacím stromem rozložen do jednotlivých stupňů linky. Každý stupeň linky potom provádí zpracování jednoho nebo více uzlů rozhodovacího stromu. Redukce paměťových nároků lze přitom dosáhnout volbou vhodných instrukcí. Novou obecnou architekturu lze použít pro jakýkoliv přístup využívající rozhodovací strom, např. HiCuts, HyperCuts nebo HyperSplit, případně další podobné algoritmy založené na rozhodovacích stromech.

Obecná řetězená linka je schematicky znázorněna na obrázku 5.1. Řetězená linka se skládá z identických výpočetních stupňů vzájemně oddělených registry, přičemž každý stupeň obsahuje paměť instrukcí a pravidel *(MEM)*, procesní element *(PE)* a multiplexor pro výběr vstupního operandu *(MUX)*. Vstupem celé linky jsou extrahované položky z hlaviček



Obrázek 5.1: *Schéma prvotního návrhu obecné řetězené linky.*



Obrázek 5.2: *Procesní element implementující instrukci algoritmu HyperSplit.*

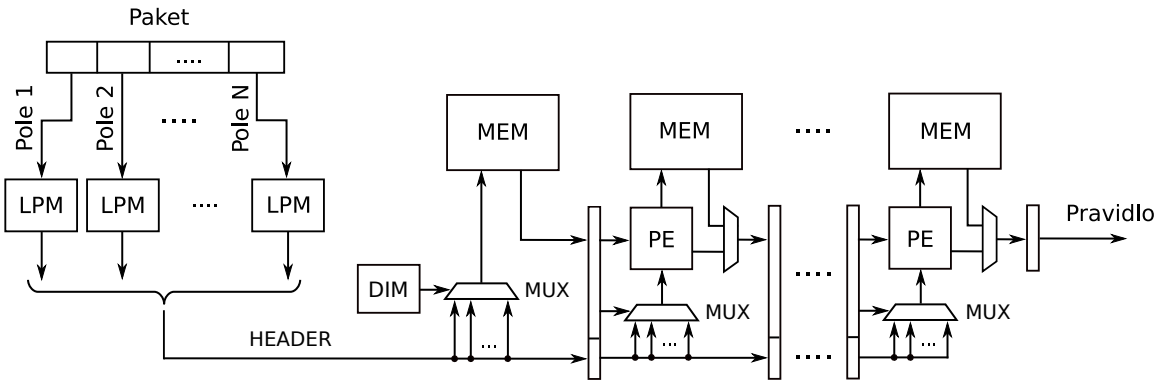
paketů *(HEADER)* a počáteční instrukce prováděného programu *(IR)*, která odpovídá kořenovému uzlu rozhodovacímu stromu. Výsledkem vykonání programu a výstupem řetězené linky je pak nalezené pravidlo, které odpovídá zase některému z listových uzlů stromu.

Zpracování instrukcí v prvním a každém dalším stupni probíhá následovně. Multiplexor vybírá vstupní operand, což je vždy jedna z položek z hlavičky paketu. Procesní element potom na základě vybrané položky (vstupního operandu) a parametrů instrukce provede výpočet adresy dalšího kroku programu – adresy následující instrukce nebo již výsledného výstupního pravidla. Je vydán požadavek na čtení z paměti a nově načtená položka je předána dalšímu stupni ke zpracování.

V případě, že je z paměti načteno již odpovídající výsledné pravidlo (nikoliv instrukce), není toto pravidlo při vstupu do dalších stupňů pipeline už nijak zpracováváno. V rámci následujících stupňů není generován požadavek na čtení z paměti a nalezené pravidlo je pouze postupně mezi stupni předáváno až k výstupu linky.

Univerzálnost linky spočívá v obecném procesním elementu, který umožňuje implementovat libovolnou funkci (instrukci). Koncept připouští implementaci i několika typů instrukcí současně. Podle typu vstupní instrukce je potom v každém stupni zvolen správný způsob zpracování procesním elementem. U konfigurovatelných technologií (např. FPGA) je možné jednoduše přizpůsobit hloubku řetězené linky konkrétní sadě pravidel pouhým přidáním dalších stupňů.

Obrázek 5.2 znázorňuje příklad procesního elementu, který implementuje instrukci algoritmu HyperSplit. V rámci instrukce je zakódována porovnávaná hodnota *(VALUE)*, bázová adresa následující instrukce *(BASE-ADDR)* a zda se jedná o reprezentaci listového uzlu *(LEAF)*. Součástí instrukce je také parametr pro výběr vstupního operandu, který však probíhá mimo procesní element a není tak na obrázku znázorněn. Při zpracování instrukce je nejprve vybrán vstupní operand a je provedeno porovnání *(CMP)* s uloženou hodnotou *(VALUE)*. Na základě bázové adresy *(BASE-ADDR)* a výsledku porovnání je potom určena adresa následující instrukce programu *(ADDR)*. Požadavek na čtení další instrukce z paměti *(READ)* je generován pouze v situaci, že se nejedná o listový uzel. Ve druhém případě je vstup procesního elementu pouze propagován na jeho výstup.



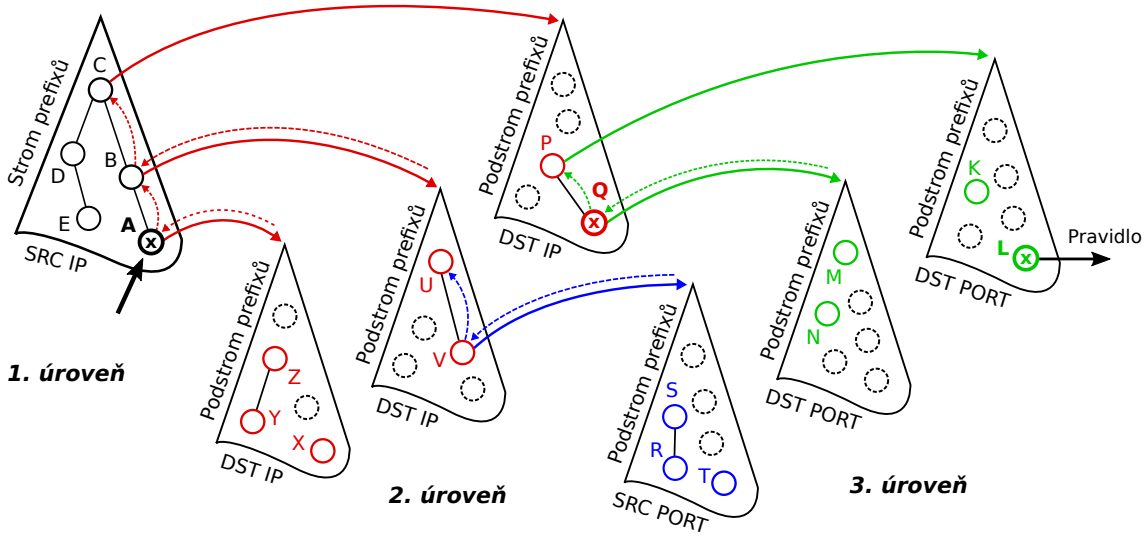
Obrázek 5.3: *Schéma modifikované řetězené linky s oddělenou klasifikací v jednotlivých dimenzích.*

5.1 Oddělení klasifikace jednotlivých dimenzí

Z analýzy klasifikačních pravidel vyplývá, že jednotlivé dimenze obsahují v porovnání s celkovým počtem pravidel datové sady vždy jen velmi malý počet unikátních hodnot a prefixů. Této vlastnosti je možné vhodně využít k optimalizaci paměťové, ale i časové složitosti procesu klasifikace. Výsledku lze dosáhnout oddělením klasifikace jednotlivých dimenzí v podobě předřazení operace LPM. To pak umožňuje řešení celého problému nad mnohem menší doménou. Vhodným příkladem pro srovnání je práce s původní hodnotou, kdy k vyjádření IPv4 adresy je potřeba 32 bitů, a úspory při použití operace LPM a práce až s výslednými prefixy, kdy k vyjádření všech možných prefixů (např. až 1024) stačí pouze 10 bitů. Použití operace LPM sice přináší určitou režii v podobě dodatečných hardwarových zdrojů, na druhou stranu to ale umožňuje právě redukci počtu bitů (až na jednu třetinu) a zase vede k výrazné úspoře zdrojů v každém stupni celé řetězené linky díky zjednodušení realizace multiplexorů a komparátorů. Při zachování stejné datové šířky instrukce to také umožňuje v rámci jedné instrukce zakódovat a provádět více porovnání najednou. Oddělením klasifikace jednotlivých dimenzí a klasifikace celé množiny pravidel tak vzniká zcela nový způsob klasifikace paketů. Schéma nové (patřičně modifikované) řetězené linky je názorně uvedeno na obrázku 5.3. Vstupem linky (na obrázku vlevo) nyní nejsou přímo položky z hlaviček paketů, ale až výsledky operace LPM nad každou dimenzí.

Naprosto odlišný je však přístup nové architektury ke konstrukci rozhodovacího stromu, respektive jemu odpovídajícího programu. Nepracuje se totiž s klasickým rozhodovacím stromem jako u algoritmu HyperSplit, nýbrž se jedná o komplikovanější několikaúrovňový průchod stromy a podstromy prefixů v jednotlivých dimenzích. Pro dosažené výraznější redukce paměťových nároků se přitom při průchodu stromem při klasifikaci využívá techniky zpětného navracení (backtrackingu). Backtracking umožňuje sdílení kódu jednotlivých větví programu. Jednotlivé segmenty programu tak nemusí být uloženy v paměti vícekrát, jsou uloženy pouze jednou. Přístup je demonstrován na příkladu z obrázku 5.4.

Příklad ukazuje postup při klasifikaci paketu, kde výsledkem předřazené operace LPM v jednotlivých dimenzích jsou následující symbolicky označené prefixy: *A (SRC IP)*, *Q (DST IP)*, ** (SRC PORT)* a *L (DST PORT)*. Průchod stromu prefixů začíná nejdelším shodným prefixem nalezeným při klasifikaci paketu v první (primární) dimenzi. V rámci příkladu na obrázku 5.4 je primární dimenzí dimenze zdrojových IP adres *(SRC IP)*. Nejdelším prefixem v této dimenzi a počátečním bodem průchodu je potom prefix označený písmenem *A* (v levé



Obrázek 5.4: *Průchod stromy a podstromy prefixů jednotlivých dimenzí při klasifikaci se zpětným navracením (backtrackingem).*

části obrázku, označeno černou tučnou šipkou). Pro prefix A jsou následně vybrány prefixy druhé (sekundární) dimenze, v tomto příkladu dimenze cílových IP adres *(DST IP)*. Jde o prefixy označené jako X, Y a Z (postup při průchodu je naznačen plnou červenou šipkou). Nejedná se o všechny prefixy z dimenze cílových adres, nýbrž o prefixy, které spolu s prefixem A z předchozí (primární) dimenze tvoří určité pravidlo, respektive odpovídají pravidlu. Tyto prefixy jsou porovnávány s výsledkem operace LPM z druhé (sekundární) dimenze. Nejdelším prefixem (výsledkem LPM) v dimenzi cílových adres je přitom prefix Q. Žádný z prefixů X, Y, Z proto při porovnání nevyhoví. V případě shody by se pokračovalo zanořením do další úrovně a porovnáním prefixů další dimenze, případně by bylo výsledkem již odpovídající pravidlo odkazované z daného prefixu. Ke shodě však nedošlo, proto se provádí backtracking (označeno červenou přerušovanou šipkou). V primární dimenzi se pak postupuje k obecnějšímu prefixu B, který je nalezenému prefixu A v této dimenzi nadřazen. Také pro prefix B jsou vybrány související prefixy U, V v sekundární dimenzi *(DST IP)*, přičemž prefix V dále odkazuje na porovnání v další dimenzi třetí úrovně (označeno šipkou modře). Ani v tomto podstromu prefixů cílových adres však nedochází ke shodě s nalezeným prefixem Q a opět se provádí backtracking. V primární dimenzi se tedy pokračuje prefixem C. V rámci sekundární dimenze už na prefixu Q dochází ke shodě a postupuje se do podstromu třetí dimenze *(DST PORT)*. Výsledkem operace LPM v dimenzi cílových portů je prefix L. V rámci podstromu, který obsahuje prefixy M, N a který je odkazován z prefixu Q z předchozí dimenze (označeno zeleně) ke shodě nedochází. Provádí se backtracking a pokračuje se nadřazeným prefixem P. V odkazovaném podstromu v dimenzi cílových portů ke shodě na prefixu L už dochází. Jelikož prefix L neobsahuje žádný odkaz do další úrovně stromu je výstupem odpovídající hledané pravidlo.

Použití backtrackingu umožňuje dosažení značné úspory paměti díky možnosti sdílení některých segmentů programu. Program odpovídající obecnějším prefixům je totiž sdílen se specifičtějšími prefixy. Na ilustraci z obrázku 5.4 je například stejný program odpovídající prefixu C prováděn také při zpětném navracení z prefixů A, B (případně D, E). Stejná situace ale nastává i u prefixů P, Q v dalších dimenzích a úrovních stromu. Na druhou stranu backtracking určitým způsobem ovlivňuje potřebnou hloubku řetězené linky (kvůli

mapování programu do pipeline). Z analýzy sad klasifikačních pravidel však vyplývá, že maximální počet prefixů odpovídající klasifikovanému paketu je velmi malý a se vzrůstajícím počtem pravidel se prakticky nemění, tzn. počet možných návratů je tak v každé dimenzi omezený.

Volba primární dimenze a pořadí dalších sekundárních dimenzí ovlivňuje výsledné parametry programu (počet uzlů rozhodovacího stromu nebo počet stupňů linky). Pořadí sekundárních dimenzí přitom nemusí být v rámci jednoho programu pevné, ale může se v jednotlivých větvích programu lišit. Například na obrázku 5.4 je ve třetí úrovni stromu v jednom případě dimenze zdrojových portů (znázorněno modře), v druhém případě dimenze cílových portů (znázorněno zeleně). Při konstrukci programu pro pipeline je potom důležité najít takovou permutaci dimenzí, která je z pohledu vlastností výsledného programu nejvýhodnější.

Navržený přístup konstrukce a průchodu stromu vyžaduje také určitou modifikaci řetězené linky. Změny nespočívají pouze v předřazení operace LPM, ale také v úpravě počáteční instrukce programu. Program totiž nezačíná vždy stejnou instrukcí, ale závisí na výsledku LPM v primární dimenzi (znázorněno také na schématu modifikované řetězené linky na obrázku 5.3). Pomocí multiplexoru je dle hodnoty v registru *DIM* (určení primární dimenze) vybrán odpovídající výsledek LPM. Na jeho základě je proveden přístup do paměti a načtena vždy správná počáteční instrukce programu.

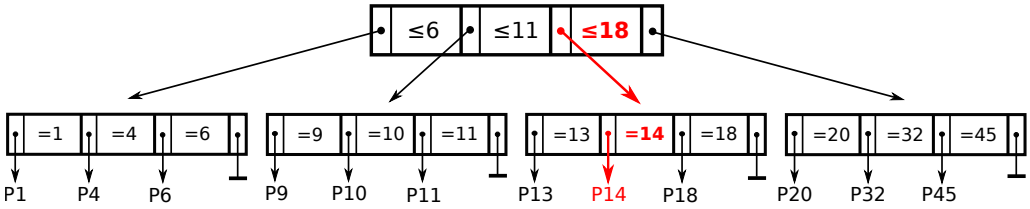
Další z potřebných modifikací řetězené linky je přidání podpory pro přeskočení určitého počtů stupňů linky při zpracování instrukce. V některých případech (například při provádění backtrackingu) je totiž nutné zajistit provedení instrukce v jiném než bezprostředně následujícím stupni linky. Tento požadavek je třeba zohlednit i při návrhu jednotlivých instrukcí. Konkrétní způsob řešení je proto popsán níže v části věnované návrhu instrukcí.

Celý koncept umožňuje také jednoduché nahrazení operace LPM jinou dílčí metodou. Například pro doménu zdrojových a cílových portů je dle analýzy sad pravidel výhodnější intervalové namísto prefixového porovnávání. Při konstrukci a průchodu stromu (programu) lze přitom bez problémů pracovat i s rozsahy. Stejně jako u prefixů mohou rozsahy tvořit hierarchickou strukturu a být vnořeny jeden do druhého (být pokryty jeden druhým). Jediným omezením je, že se rozsahy nesmí vzájemně ani částečně překrývat. Z důvodu provádění backtrackingu musí být pro daný rozsah vždy jednoznačně určeno, který z dalších rozsahů je mu přímo nadřazen. Z analýzy sad klasifikačních pravidel však vyplývá, že nežádoucí překrytí rozsahů, se vyskytuje jen velmi zřídka (v jednotkách případů). Jen tehdy je nutné daný interval rozdělit na dva podintervaly. Náhrada operace LPM u portů umožňuje odstranit potřebu převádět rozsahy na prefixy, čímž se lze vyhnout nežádoucí expanzi počtu pravidel.

Pro hledání shody prefixu v jednotlivých krocích programu lze volit dva možné přístupy, n-ární porovnání nebo hashovací tabulku. Z těchto vyhledávacích datových struktur vychází také návrh dvojice instrukcí pro řetězenou linku.

5.1.1 Instrukce využívající n-ární porovnání

První z dvojice instrukcí je založena na vyhledávacím stromu. Návrh instrukce vychází z principu algoritmu HyperSplit, který používá binární vyhledávací strukturu. Na rozdíl od HyperSplit pracuje nově navržená instrukce nad výrazně menší doménou hodnot a umožňuje tak provádět až n porovnání v jednom kroku. Vyhledání se provádí nad seřazenou posloupností. V každém kroku je porovnáno n hodnot a zjištěno, ve které z $(n+1)$ částí posloupnosti se může položka nacházet. Princip n-árního stromového vyhledávání ukazuje obrázek 5.5.



Obrázek 5.5: *Princip n-árního stromového vyhledávání (vyznačen příklad vyhledání položky s hodnotou 14).*

Kořenový uzel obsahuje tři hodnoty a čtyři ukazatele. Význam ukazatelů a příslušných hodnot je přitom následující. První ukazatel odkazuje na podstrom obsahující pouze položky s indexem i, kde $i \leq 6$. Druhý odkazuje na podstrom s položkami $6 < i \leq 11$ a třetí odpovídá položkám $11 < i \leq 18$. Poslední ukazatel odkazuje na podstrom s položkami $i > 18$. Při hledávání položky s hodnotou 14 (znázorněno červeně) je proto zvolen třetí z podstromů. V rámci poslední (listové) úrovně stromu jsou potom porovnávány přímo indexy uložených položek. Příslušné ukazatele pak rovnou odkazují na hledané položky (např. *P14*). Stejný princip vyhledávání byl uplatněn také při návrhu instrukce. Prohledávané prefixy jsou seřazeny do posloupnosti a každá instrukce umožňuje současné porovnání až tří z nich a určení následujícího postupu při provádění programu.

Obrázek 5.6 znázorňuje schéma možné podoby procesního elementu. V rámci každé instrukce jsou zakódovány porovnávané hodnoty *(VALUE1-3)* a bázová adresa následující instrukce *(BASE-ADDR)*. Podle výsledků porovnání trojice hodnot se vstupním operandem *(CMP)* je odvozena konkrétní adresa *(ADDR)* jedné ze čtyř následujících instrukcí programu. Požadavek na čtení z paměti *(READ)* není generován v případě, kdy je vstupem již výsledné pravidlo *(LEAF)* nebo v případě, kdy je třeba pro zpracování instrukce přeskočit několik stupňů linky, tj. je-li položka *SKIP* v instrukci nenulová. Hodnota *SKIP* přímo udává počet přeskakovaných stupňů linky. Taková instrukce je potom předána následujícímu procesnímu elementu a současně je u ní dekrementována hodnota položky *SKIP*.

Konkrétní formát a zvolené zakódování instrukce ukazuje tabulka 5.1. Položka *leaf* slouží k určení, zda se jedná o výsledné pravidlo (hodnota 1) nebo instrukci určenou ke zpracování (hodnota 0), *opcode* určuje typ instrukce, položka *operand* slouží k výběru vstupního operandu a *skip* vyjadřuje počet přeskakovaných stupňů linky. Položky *value1-3* jsou porovnávané hodnoty a *base-addr* určuje bázovou adresu následující instrukce programu.

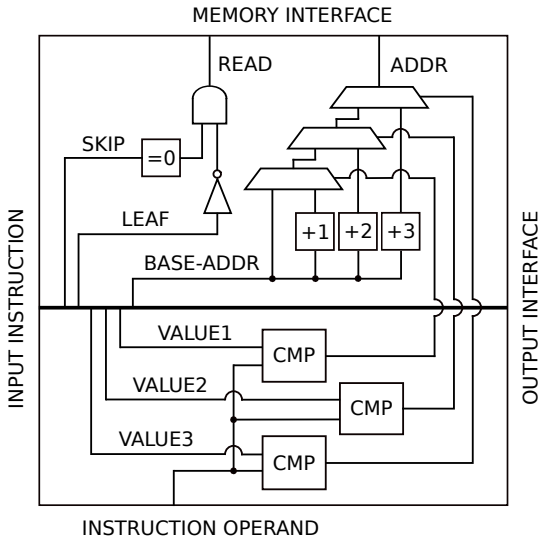
Tabulka 5.1: *Možný formát a zakódování instrukce využívající n-ární porovnání.*

leaf	***opcode***	***operand***	***skip***	***value1***	***value2***	***value3***	***base-addr***
1 bit	3 bity	4 bity	5 bitů	13 bitů	13 bitů	13 bitů	20 bitů

Pro úplnost je v tabulce 5.2 uveden také formát ukládaného pravidla. Položka *leaf* opět slouží k určení, zda se jedná o pravidlo (hodnota 1) nebo instrukci určenou ke zpracování (hodnota 0). Pole *rule-data* pak obsahuje uložené informace k nalezenému pravidlu.

Tabulka 5.2: *Možný formát a zakódování pravidla.*

leaf	***rule-data***
1 bit	71 bitů



Obrázek 5.6: *Procesní element implementující instrukci využívající n-ární porovnání.*

5.1.2 Instrukce využívající hashovací funkci

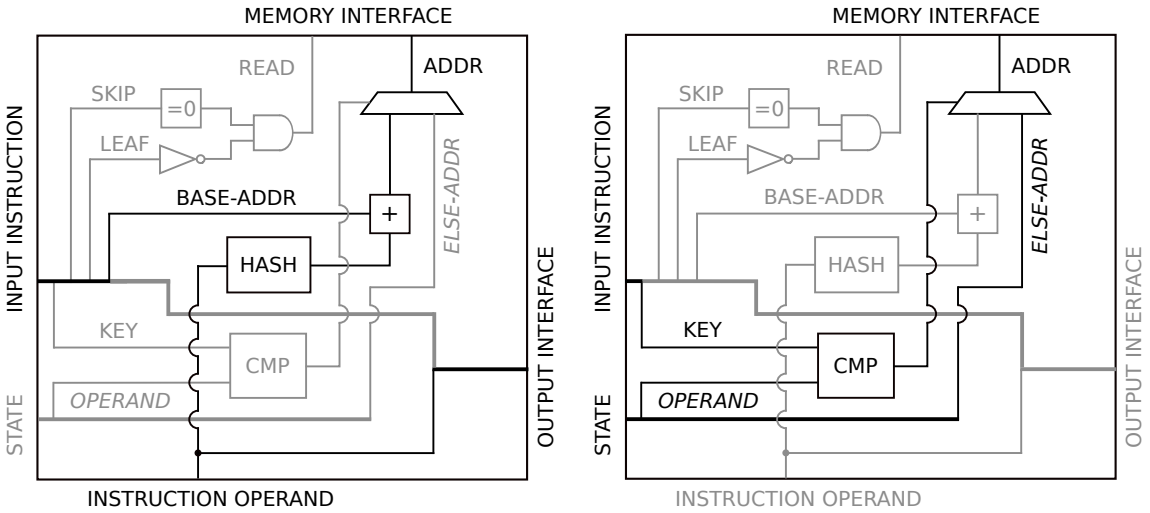
Druhá z instrukcí je založena na principu hashovací tabulky. Cílem při návrhu bylo umožnit porovnávání ještě více hodnot v jediném kroku a zajistit tak snížení hloubky pipeline i počtu uzlů programu.

Formát a způsob zakódování instrukce ukazuje tabulka 5.3. Několik prvních položek je zcela shodných s předchozí instrukcí. Položka *leaf* opět určuje, zda se jedná o instrukci (hodnota 0) nebo pravidlo (hodnota 1). Pole *opcode* určuje typ instrukce, položka *operand* slouží k výběru vstupního operandu instrukce a *skip* vyjadřuje počet přeskakovaných stupňů linky. Z důvodu možných kolizí při použití hashovací tabulky je nutné do paměti ukládat také hodnotu klíče, aby bylo možné při hledání ověřit správnost nalezeného výsledku. Položka *key* v instrukci vyjadřuje právě hodnotu klíče. Pole *base-addr* vyjadřuje bázovou adresu pro přístup do paměti a *else-addr* obsahuje adresu alternativní větve programu, kterou se pokračuje v případě, že položka s daným klíčem nebude nalezena. Poslední položka *reserved* je rezervována pro budoucí použití. V případě potřeby umožňuje rozšíření některého z předchozích polí.

Tabulka 5.3: *Možný formát a zakódování instrukce využívající hashovací funkci.*

leaf	***opcode***	***operand***	***skip***	***key***	***base-addr***	***else-addr***	***reserved***
1 bit	3 bity	4 bity	5 bitů	13 bitů	20 bitů	20 bitů	6 bitů

Vyhledávání založené na principu hashovací tabulky pracuje ve dvou krocích a zpracování každé instrukce tak musí probíhat ve dvou po sobě jdoucích procesních elementech. Z tohoto důvodu je také nutné umožnit při implementaci předávání určité stavové informace mezi jednotlivými stupni linky. Obrázek 5.7 zobrazuje schéma procesního elementu s vyznačením obou fází zpracování instrukce. V prvním kroku (na obrázku vlevo) je na základě vstupního operandu, odpovídajícího výsledku hashovací funkce *(HASH)* a bázové adresy *(BASE-ADDR)* určena možná pozice hledané položky v paměti *(ADDR)*. Je proveden přístup do paměti. V druhém kroku (na obrázku vpravo) v následujícím procesním elementu se potom provádí ověření výsledku porovnáním *(CMP)* hodnoty klíče *(KEY)* a operandu



Obrázek 5.7: *Procesní element implementující instrukci využívající hashovací funkci.*

z předchozího stupně *(OPERAND)*. Každý procesní element tak souběžně provádí ověření správnosti načtené vstupní instrukce a výpočet adresy instrukce potenciálně následující. V případě nerovnosti klíčů (položka nebyla nalezena) je při přístupu do paměti vnucena adresa alternativní větve programu *(ELSE-ADDR)* získaná z předchozího stupně linky. Mechanismus přeskakování stupňů linky *(SKIP)* a detekce nalezeného pravidla *(LEAF)* pracují stejným způsobem jako u předchozí instrukce.

Pro doplnění je uvedena také tabulka 5.4 obsahující formát ukládaného pravidla. Položka *leaf* určuje, zda se jedná o pravidlo (hodnota 1) nebo instrukci (hodnota 0). Stejně jako instrukce i pravidlo musí obsahovat hodnotu klíče *(key)*, aby bylo možné ověřit správnost nalezené položky z paměti. Pole *rule-data* pak obsahuje uložené informace k nalezenému pravidlu.

Tabulka 5.4: *Možný formát a zakódování pravidla.*

leaf	***key***	***rule-data***
1 bit	13 bit	58 bitů

5.2 Konstrukce programu pro pipeline

Nově navržená architektura s předřazenou operací LPM vyžaduje odlišný přístup ke konstrukci rozhodovacího stromu. Z tohoto důvodu bylo nutné vytvořit také zcela nový algoritmus pro transformaci vstupní sady klasifikačních pravidel na program a jeho následné mapování do pipeline. Základem celého přístupu konstrukce programu je rekurzivní funkce *procRuleset()*, která provádí převod sady pravidel do pomocné reprezentace, graf závislostí. Implementace funkce v podobě pseudokódu je uvedena v rámci bloku algoritmus 5.1.

Vstupem funkce je množina pravidel *(rules)* a množina dimenzí datové sady *(dims)*. Na počátku funkce jsou nejprve uvedeny ukončovací podmínky umožňující přerušení rekurzivního volání. V případě, že je množina pravidel *(rules)* prázdná, vrací funkce prázdnou hodnotu *None* (řádky 2-3). V případě, že je množina dimenzí *(dims)* prázdná, vrací funkce nový listový uzel *LeafNode* reprezentující první pravidlo z množiny pravidel *(rules[0])*, které

Algoritmus 5.1: *Rekurzivní funkce* procRuleset() *pro konstrukci grafu závislostí.*

```
1  function procRuleset(rules, dims):
2      if (rules is Empty):
3          return None
4
5      if (dims is Empty):
6          return new LeafNode(rules[0])
7
8      outNode ← None
9
10     for (dim in dims):
11         currNode ← new Node(dim, rules)
12         prefs ← getDimPrefs(dim, rules)
13         nextDims ← dims \ {dim}
14
15         for (pref in sort(prefs)):
16             nextRules ← getPrefRules(pref, rules)
17
18             prefParent ← pref.parent
19             while (prefParent is not None):
20                 nextRules ← nextRules \ currNode.childs[prefParent].rules
21                 prefParent ← prefParent.parent
22
23             currNode.childs[pref] ← procRuleset(nextRules, nextDims)
24
25         if (outNode is None or cost(currNode) ≤ cost(outNode)):
26             outNode ← currNode
27
28     return outNode
```

odpovídá právě tomu s nejvyšší prioritou (řádky 5-6). V rámci hlavního těla funkce je nejprve inicializována hodnota proměnné *outNode* reprezentující aktuální kandidátní a výstupní uzel funkce (řádek 8). Následně je prováděn průchod přes všechny prvky množiny dimenzí (řádek 10). Pro každou dimenzi *dim* je vytvořen nový potenciálně kandidátní uzel *currNode* konstruovaného grafu (řádek 11), jsou získány všechny prefixy *prefs* odpovídající vstupní množině pravidel *(rules)* v dané dimenzi (volání funkce *getDimPrefs()*, řádek 12) a je vytvořena nová množina dimenzí *nextDims*, která obsahuje všechny dimenze původní vstupní množiny *dims* kromě té aktuálně zpracovávané (řádek 13). Symbolem \ je označena operace množinového rozdílu. Následně je prováděn průchod přes všechny získané prefixy *prefs* (řádek 15). Prefixy jsou při průchodu přitom seřazeny tak, aby kratší a obecnější prefixy byly zpracovány dříve (funkce *sort()*). Pro každý prefix *pref* je získána podmnožina *nextRules* původní množiny pravidel *rules*, kterým v dané dimenzi *dim* odpovídá prefix *pref* (volání funkce *getPrefRules()*, řádek 16). Na řádcích 18-21 funkce je provedeno odstranění všech pravidel z množiny *nextRules*, která jsou už pokryta obecnějšími a nadřazenými prefixy právě zpracovávaného prefixu *pref*. Notace *pref.parent* přitom vyjadřuje právě přístup k přímo nadřazenému rodičovskému prefixu. Následně je provedeno rekurzivní volání

Tabulka 5.5: *Sada pravidel použitá v ukázkovém příkladu konstrukce programu a přehled prefixů z jednotlivých dimenzích spolu s označením, kterým pravidlům prefixy odpovídají.*

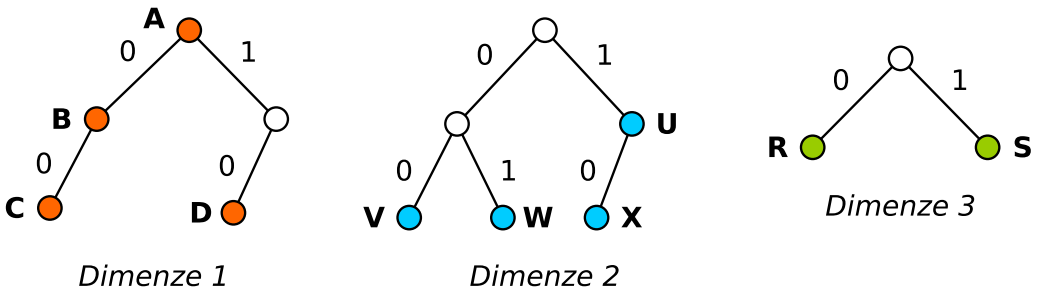
	Dim 1	Dim 2	Dim 3
P_0	00∗ *(C)*	1∗ *(U)*	0∗ *(R)*
P_1	0∗ *(B)*	10∗ *(X)*	0∗ *(R)*
P_2	0∗ *(B)*	01∗ *(W)*	1∗ *(S)*
P_3	0∗ *(B)*	1∗ *(U)*	1∗ *(S)*
P_4	10∗ *(D)*	1∗ *(U)*	0∗ *(R)*
P_5	∗ *(A)*	00∗ *(V)*	1∗ *(S)*

	Dim 1				Dim 2				Dim 3	
	A	B	C	D	U	V	W	X	R	S
P_0			×		×			×	×	
P_1		×	×					×	×	
P_2		×	×				×			×
P_3		×	×		×			×		×
P_4				×	×			×	×	
P_5	×	×	×	×		×				×

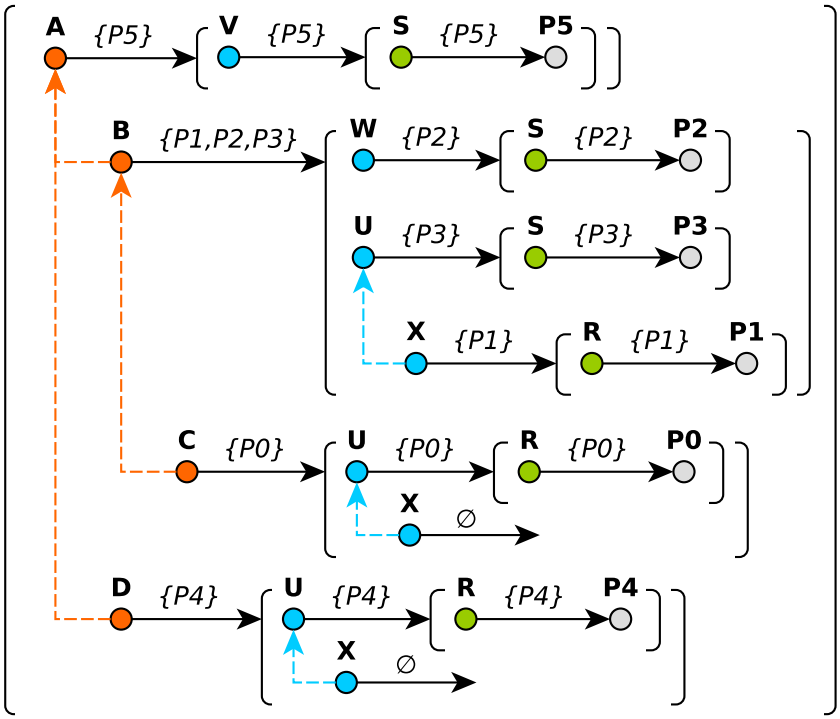
funkce nad nově získanými množinami *nextRules* a *nextDims*. Návratová hodnota funkce reprezentující uzel konstruovaného grafu pro daný prefix *pref* je potom přidána do množiny následníků aktuálně zpracovávaného uzlu *currNode* (řádek 23). Po zpracování všech prefixů je dále vyhodnocena cenová (fitness) funkce *cost()*, jednak pro současně zpracovávaný uzel *currNode*, dále pak pro aktuální kandidátní uzel *outNode* a v případě dosažení lepšího výsledku je současný kandidátní uzel aktualizován (řádky 25-26). Po zpracování všech dimenzí vrací funkce výstupní uzel *outNode* s nejnižší hodnotou cenové funkce (řádek 28).

Rekurzivní funkce tak provádí prozkoumání všech možných permutací primární a dalších sekundárních dimenzí s cílem vytvoření nejvýhodnějšího grafu závislostí a jemu odpovídajícího programu minimalizujícího cenovou funkci. V rámci cenové funkce může být při konstrukci grafu zohledněna a optimalizována paměťová či časová složitost vznikajícího programu, která odpovídá počtu uzlů, respektive potřebné hloubce řetězené linky.

Výsledek konstrukce grafu závislostí je názorně demonstrován na příkladu jednoduché sady klasifikačních pravidel. Všechna vstupní pravidla jsou uvedena v tabulce 5.5 (vlevo). Řádky tabulky odpovídají pravidlům seřazeným podle priority (pravidlo P_0 s nejvyšší prioritou až P_5 s nejnižší prioritou). V tabulce jsou uvedeny prefixy každého pravidla pro jednotlivé dimenze. Pro lepší přehlednost a orientaci jsou prefixy symbolicky označeny písmeny velké abecedy. Pravá část tabulky 5.5 navíc poskytuje přehled prefixů ze všech dimenzí spolu s označením, kterým pravidlům jednotlivé prefixy odpovídají (označeno křížkem ×). Taková tabulka je velmi užitečnou pomůckou při konstrukci grafu závislostí. Tabulka 5.5 je pro názornost ještě doplněna obrázkem 5.8, který ukazuje hierarchii prefixů v jednotlivých dimenzích. Pro dimenzi 1 (vlevo) jsou prefixy znázorněny červeně, pro dimenzi 2 (uprostřed) modře a pro dimenzi 3 (vpravo) zeleně.



Obrázek 5.8: *Znázornění hierarchie prefixů v ukázkovém příkladu konstrukce programu.*

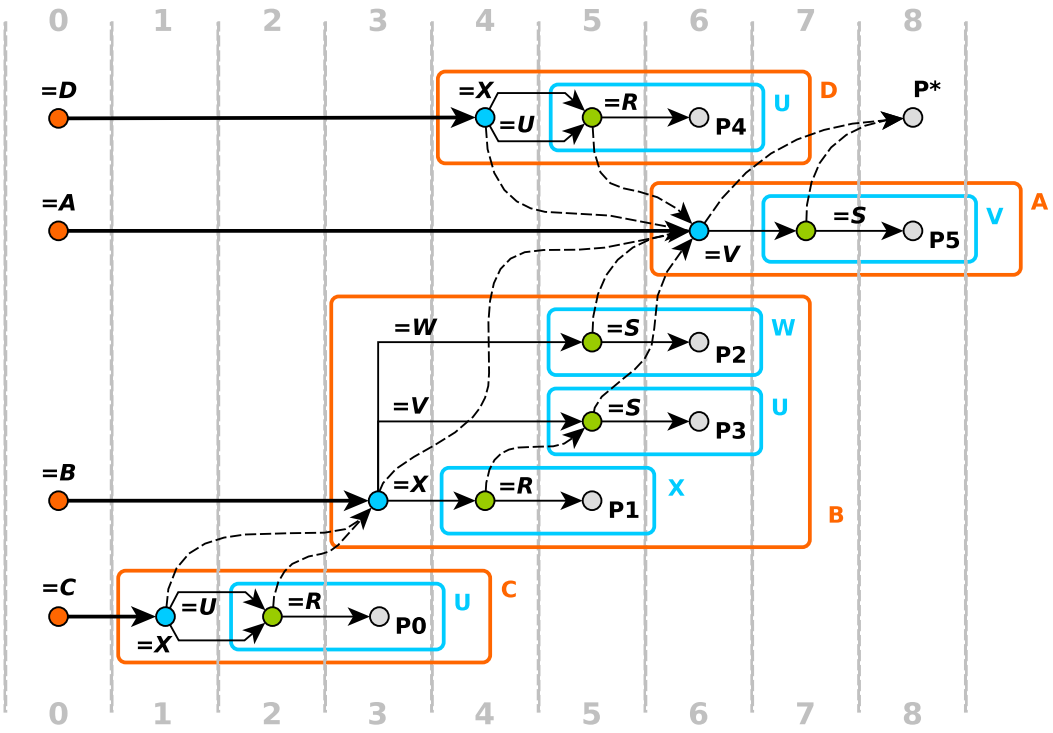


Obrázek 5.9: *Graf závislostí vytvořený na základě sady klasifikačních pravidel.*

Vytvořený graf závislostí je uveden na obrázku 5.9. Pro jednoduchost nebyly při konstrukci prozkoumávány všechny permutace, ale uvažováno pouze pevné pořadí dimenzí (v pořadí dimenze 1, dimenze 2, dimenze 3). Výsledný graf kopíruje hierarchii prefixů z obrázku 5.8 (závislosti prefixů jsou znázorněny přerušovanou čarou). Graf zároveň přehledně ukazuje průchod podstromy prefixů následujících sekundárních dimenzí při klasifikaci. Jednotlivé prefixy jsou opět značeny stejnými barvami (podle dimenzí) jako na předchozím obrázku (dimenze 1 červeně, dimenze 2 modře, dimenze 3 zeleně). Šedou barvu mají listové uzly představující jednotlivá pravidla. Velké závorky přitom ohraničují jednotlivá rekurzivní volání funkce *procRuleset()*. Skupina prefixů stejné barvy v rámci volání funkce představuje potomky uzlu předchozí dimenze. Nad šipkami znázorňujícími průchod přes dimenze jsou navíc uvedeny vstupní množiny pravidel každého volání.

Takto vytvořený graf závislostí je hned v následující fázi konstrukce programu mapován s použitím navržených instrukcí do jednotlivých stupňů řetězené linky. Výsledek mapování grafu z obrázku 5.9 při použití instrukce n-árního porovnání pro dříve uvedenou sadu klasifikačních pravidel je znázorněn na obrázku 5.10.

Svislými šedými čarami s číselným označením jsou na obrázku znázorněny jednotlivé stupně řetězené linky. Uzly zobrazeného grafu tentokrát nepředstavují prefixy dimenzí, nýbrž samotné instrukce programu. Porovnávané prefixy v rámci určité instrukce jsou vyznačeny na hranách grafu příslušejících danému uzlu. Použité instrukce přitom umožňují porovnání až tří prefixů současně. Přerušovanou čarou jsou potom znázorněny alternativní větve programu, které odpovídají také prováděnému zpětnému navracení (backtrackingu) rozprostřenému přes stupně pipeline. Alternativní větev je vykonávána v případě, že v přímé větvi programu nedochází ke shodě prefixu. Barevnými obdélníky jsou orámovány sekvence

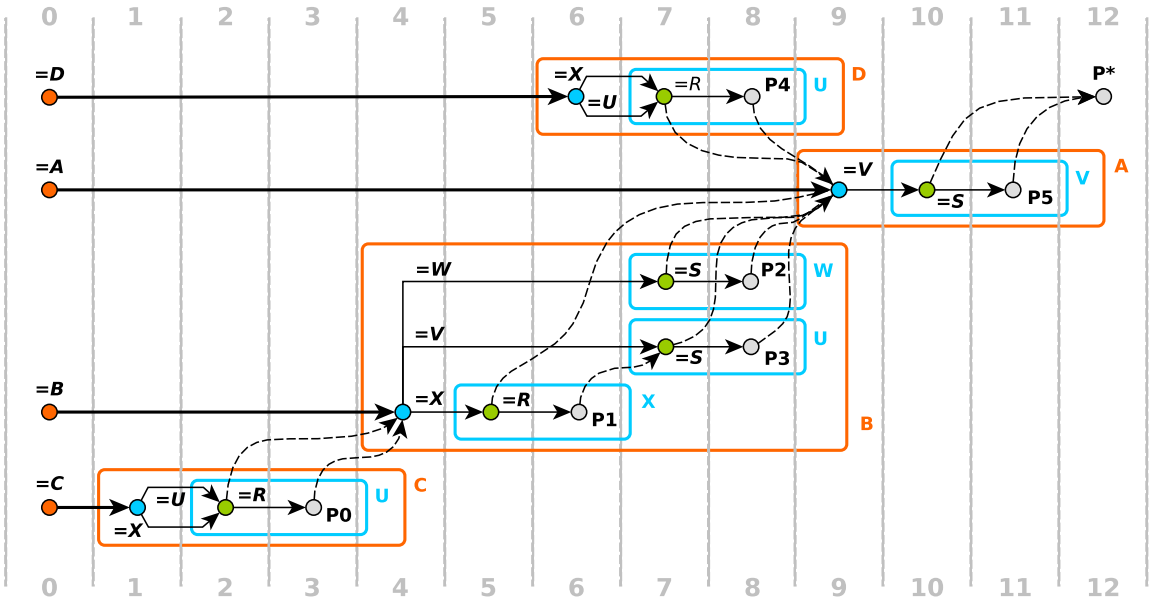


Obrázek 5.10: *Mapování programu do pipeline s využitím instrukce n-ární porovnání.*

instrukcí (segmenty programu) příslušející prefixům primární (červeně), respektive sekundární dimenze (modře). V nultém stupni řetězené linky jsou červenými uzly znázorněny počáteční instrukce programu, které odpovídají a závisí na výsledku operace LPM v primární dimenzi. Šedou barvou jsou potom znázorněny uzly představující výsledná pravidla procesu klasifikace. Pravidlo $P*$ značí výchozí pravidlo, které je výstupem klasifikace v případě, že daný paket neodpovídá žádnému z pravidel datové sady.

U příkladu zvolený postup plánování programu do pipeline odpovídá principu ALAP *(As Late As Possible)*, tzn. instrukce je vždy mapována do nejvzdálenějšího možného stupně řetězené linky. Nejprve jsou plánovány uzly, které nemají žádné závislosti. To odpovídá nejobecnějším prefixům. Následně jsou plánovány ty uzly, jejichž všichni následníci byli již v některém z předchozích kroků úspěšně naplánováni, tzn. prefixy B, D jsou plánovány až po umístění prefixu A, stejně tak prefix C je zpracován až po naplánování prefixu B. Algoritmus pokračuje stejným způsobem, dokud nejsou naplánovány všechny uzly. Z důvodu dříve pevně zvoleného pořadí dimenzí nemusí být výsledný plán programu na příkladu z obrázku 5.10 z pohledu potřebné hloubky řetězené linky optimální.

Pro úplnost je na obrázku 5.11 znázorněn také výsledek mapování stejného programu do pipeline při použití instrukce využívající hashování. Na první pohled je zřejmé, že instrukce s hashováním dosahuje na příkladu horších výsledků. Důvodem je odlišný způsob vyhodnocování jednotlivých instrukcí, kdy každá instrukce musí být zpracována vždy ve dvou po sobě jdoucích stupních řetězené linky. Alternativní větev programu (přerušovaně) tak musí být plánována až do dalšího stupně, což zvyšuje potřebnou hloubku řetězené linky. Mohlo by se zdát, že z tohoto důvodu nedává použití takové instrukce příliš smysl. Výhody instrukce se však projeví až při porovnávání mnohem většího počtu prefixů, jak bude vidět v kapitole 7.



Obrázek 5.11: *Mapování programu do pipeline s využitím instrukce s hashováním.*

Výše představné fáze vytváření programu (konstrukce grafu závislostí a mapování do pipeline) musí ve skutečnosti probíhat souběžně. Při hledání nejvýhodnější permutace dimenzí totiž cenová funkce zohledňuje také počet uzlů a potřebnou hloubku řetězené linky, které přímo vychází z konkrétního mapování programu do pipeline.

Kapitola 6

Hardwarová platforma COMBO

Hardwarová platforma COMBO se skládá z řady FPGA akceleračních karet určených pro připojení do PCI-Express slotu, které jsou vyvíjeny v rámci sdružení CESNET [10]. Karty slouží především pro akceleraci zpracování síťových dat na vysokorychlostních linkách, neboť disponují také optickým síťovým rozhraním. Následující části kapitoly jsou věnovány stručné charakteristice jedné z nejnovějších akceleračních karet COMBO-100G podporující technologii 100G Ethernet a vývojovému prostředí NetCOPE.

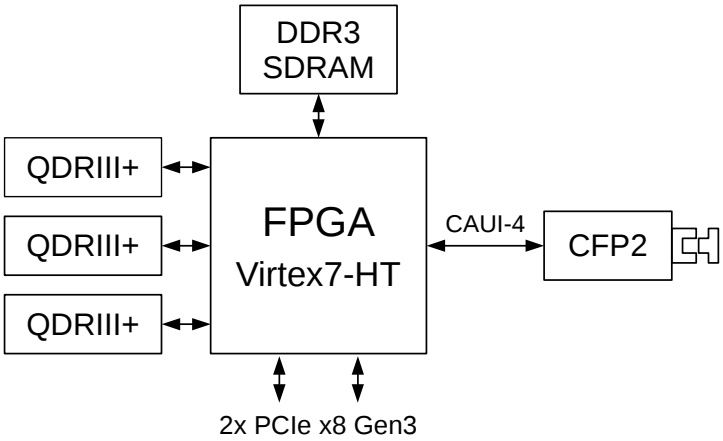
6.1 Akcelerační karta COMBO-100G

Základní blokové schéma akcelerační karty COMBO-100G je uvedeno na obrázku 6.1. Jádrem karty je čip FPGA Xilinx Virtex-7HT. Dalšími komponentami jsou optické síťové rozhraní CFP2, rozhraní PCI-Express Gen3, tři moduly statických pamětí QDRIIIe a skupina modulů dynamických pamětí DDR3 SDRAM [7]. Jednotlivé komponenty jsou dále detailněji popsány.

FPGA Virtex-7HT. FPGA čip Virtex-7 XC7VH580T [24] od společnosti Xilinx tvoří jádro karty. Základními prvky technologie FPGA jsou konfigurovatelné logické bloky (CLB). Každý blok se skládá ze dvou menších buněk (Slice). Ty jsou dále tvořeny čtyřmi 6-vstupými vyhledávacími tabulkami (LUT), osmi registry (flip-flop) a další pomocnou logikou pro realizaci rychlých aritmetických přenosů (Carry Logic). Mezi další specializované bloky patří blokové paměti (BRAM), aritmeticko-logické bloky (DSP) a vestavěné bloky pro podporu rozhraní PCI-Express a rychlých sériových spojů (GTZ/GTH Transceivers).

Rozhraní CFP2. Pro podporu technologie 100G Ethernet je na kartě použita druhá generace optického rozhraní CFP, které podporuje dva módy optického přenosu 10x10G a 4x25G. Moduly CFP2 provádí převod mezi elektrickými a optickými signály a kromě podpory 100G Ethernetu umožňují běh také v režimu deseti nezávislých 10G linek. Odpovídající elektrické rozhraní CAUI-4 je přímo připojeno k čipu FPGA prostřednictvím čtyř GTZ transceiverů. Celé propojení je tvořeno osmi diferenciálními páry vodičů (čtyři páry v každém směru), kde každý pár přenáší data rychlostí 25 Gb/s.

Rozhraní PCI-Express. Rozhraní PCI-Express slouží k přenosu síťových dat do paměti hostitelského počítače a zpět. Čip FPGA obsahuje dva vestavěné bloky pro podporu rozhraní PCI-Express Gen3 x8 s propustností 8 Gb/s pro jednu linku. Pro dosažení



Obrázek 6.1: *Základní blokové schéma karty COMBO-100G.*

celkové propustnosti alespoň 100 Gb/s jsou propojeny oba PCIe Gen3 x8 bloky do jednoho PCIe Gen3 x16 rozhraní s teoretickou propustností až 128 Gb/s. K tomuto účelu se používá technologie tzv. PCIe bifurkace, která umožňuje v rámci vhodné konfigurace základní desky a procesoru využití jediného fyzického PCIe x16 slotu. V rámci operačního systému se potom karta zařízení jeví jako dvě logická zařízení.

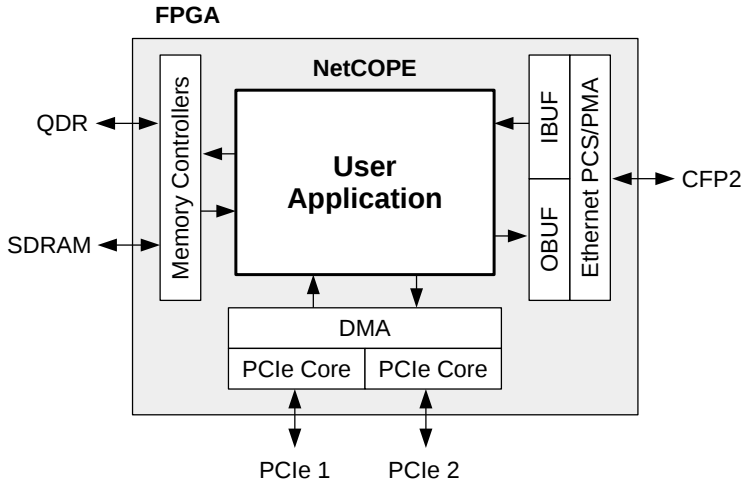
Paměť QDRIIIe. Statické paměti QDRIIIe se vyznačují nízkou latencí a vysokou propustností s náhodným přístupem. Osazené paměťové moduly poskytují (každý) dva současně použitelné porty, jeden pro čtení, druhý pro zápis. Datové přenosy mohou probíhat až na frekvenci 700 MHz DDR, přičemž šířka datové sběrnice je 36 b. To představuje propustnost více než 50 Gb/s na modul. Kapacita jednoho modulu je přitom 72 Mb a jsou použity tři moduly (celkově 216 Mb).

Paměť DDR3 SDRAM. Pro uchování velkého objemu dat s vyšší latencí při přístupu je dále k dispozici osm modulů dynamické paměti DDR3 SDRAM. Každý modul při datové šířce 8 b a frekvenci 800 MHz poskytuje propustnost 12,8 Gb/s. Kapacita jednoho modulu je 1 Gb. Použitím osmi modulů dostáváme celkovou propustnost přes 100 Gb/s s celkovou kapacitou paměti 8 Gb.

6.2 Vývojové prostředí NetCOPE

Platforma NetCOPE (Network COMBO Pipe) [11] je modulární vývojové prostředí, které poskytuje infrastrukturu, řadu knihovních komponent a softwarových nástrojů pro podporu a urychlení vývoje síťových aplikací na akceleračních kartách využívajících technologii FPGA. Bylo navrženo především pro rodinu akceleračních karet COMBO. NetCOPE si klade za cíl odstínit návrháře uživatelské aplikace od nízkoúrovňové práce s jednotlivými hardwarovými prvky konkrétní síťové karty, pro kterou je aplikace vyvíjena. Návrhář se tak nemusí soustředit na komplikovanou implementaci řady komunikačních rozhraní a může se zaměřit na tvorbu samotné aplikace.

Základní struktura firmwarové části architektury NetCOPE je zobrazena na obrázku 6.2. Mezi klíčové prvky architektury patří vstupní a výstupní síťové bloky (IBUF, OBUF, Ethernet PCS/PMA) pro podporu příjmu a vysílání dat na síťovém rozhraní Ethernet



Obrázek 6.2: *Základní blokové schéma firmwarové části platformy NetCOPE.*

(vpravo na obrázku). Dále zahrnuje řadiče externích pamětí (Memory Controllers) abstrahující přístup ke statickým a dynamickým pamětem na kartě (vlevo na obrázku). Nedílnou součástí je také zajištění komunikace po sběrnici PCI-Express a řadič přímého přístupu do operační paměti (Direct Memory Access, DMA) umožňující rychlé přenosy dat do operační paměti hostitelského počítače (v dolní části obrázku) spolu s řadou softwarových nástrojů a knihoven pro jejich správu. Dále jsou poskytovány základní komponenty pro tvorbu uživatelské aplikace např. implementace pamětí FIFO (First In First Out), pamětí CAM (Content-addressable Memory), komponent pro práci s komunikačními protokoly infrastruktury nebo bloků pro zpracování síťového provozu (extrakce položek z hlaviček paketů, výpočet kontrolního součtu CRC) a další.

Kapitola 7

Dosažené výsledky

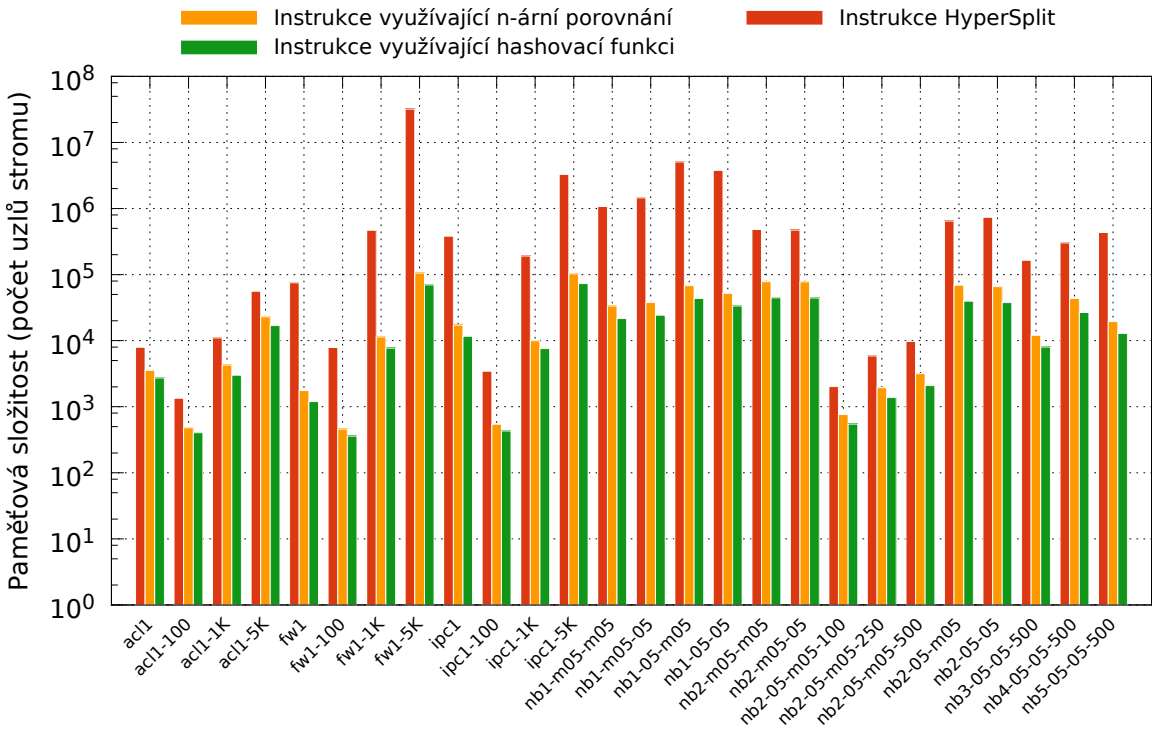
Architekturu, jejíž návrh byl popsán v předcházející kapitole 5, jsem implementoval v jazyce VHDL. Při implementaci jsem zohlednil specifika mapování do technologie FPGA a zároveň použil některé komponenty dostupné v rámci vývojové platformy NetCOPE. Šlo především o komponenty usnadňující využití blokových pamětí a distribuované paměti dostupné na čipu FPGA. Při realizaci jsem použil také hotový modul implementující operaci LPM.

Vytvořená implementace řetězené linky je do značné míry parametrizovatelná. Kromě výběru použité instrukce, nastavení hloubky řetězené linky a počtu dimenzí lze volit také množství alokované paměti a to jednotlivě zvlášť pro každý stupeň. Nastavitelný je i parametr udávající maximální množství unikátních hodnot v každé dimenzi, což určuje potřebnou kapacitu operace LPM (minimální garantovaný počet prefixů, které lze uložit). Od toho se také automaticky odvíjí datová šířka jednotlivých signálů a příslušných položek použitých instrukcí programu.

V rámci implementace byl s použitím knihovny Netbench vytvořen také Python skript, který umožňuje vyhodnocení vlastností a výsledků navrženého přístupu pro konkrétní sady klasifikačních pravidel. Zjišťována byla především paměťová složitost a potřebná hloubka řetězené linky. Pro zhodnocení parametrů nového algoritmu byly použity stejné sady filtračních pravidel jako při předchozí prováděné analýze známých a dříve představených algoritmů pro klasifikaci. Charakteristika významných vlastností použitých datových sad s filtračními pravidly byla popsána dříve v kapitole 4.

Dosažené výsledky vlastního přístupu jsou v této kapitole srovnávány s výsledky algoritmu HyperSplit. Ten totiž při předchozí analýze vykazoval vynikající poměr mezi časovou a paměťovou složitostí a jevil se pro filtraci paketů ve 100 Gb sítích jako nejvýhodnější. Je třeba zmínit, že pro korektní srovnání obou přístupů bylo nutno algoritmus HyperSplit nastavit do režimu $binth = 1$, kdy se neprovádí sekvenční dohledávání pravidel v lineárním seznamu odkazovaném z listových uzlů rozhodovacího stromu a výsledné odpovídající pravidlo je vždy nalezeno pouhým průchodem rozhodovacího stromu. U vlastního přístupu je potom při konstrukci programu pro řetězenou linku vždy hledána taková permutace dimenzí, která je z pohledu vlastností výsledného programu nejvýhodnější. Při konstrukci rozhodovacího stromu se v dimenzi zdrojových a cílových portů rovněž pracuje přímo s rozsahy, čímž se lze výhodně vyhnout nežádoucí expanzi počtu pravidel.

Požadavkem na vysokou propustnost navrženého řešení se rozumí podpora zpracování příchozích síťových dat na plné rychlosti linky, až 100 Gb/s. U nejkratších ethernetových rámců s velikostí 64 bytů to znamená při dané rychlosti zpracovat a klasifikovat až 148 809 523 paketů za sekundu. Na klasifikaci jednoho paketu potom připadá přibližně 5 nanosekund, což vyžaduje dosažení minimální pracovní frekvence 200 MHz a představuje



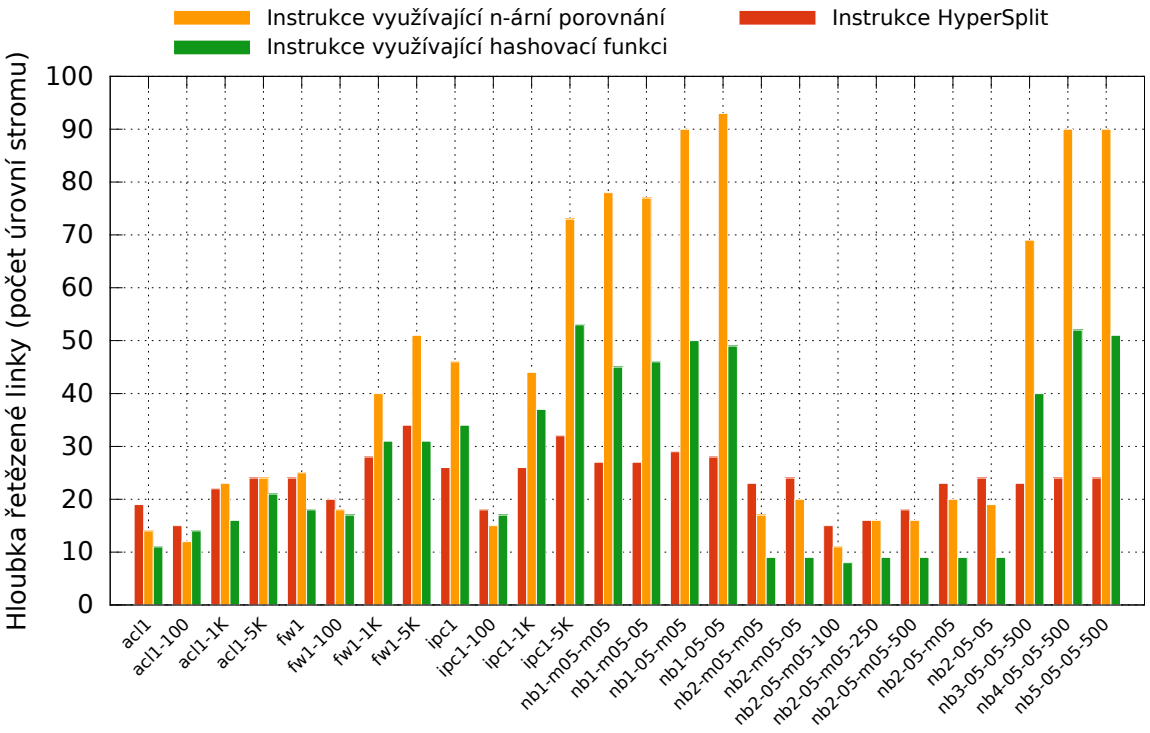
Obrázek 7.1: *Paměťová složitost (počet uzlů rozhodovacího stromu).*

možný příchod paketu v každém taktu hodinového signálu a s tím také nutnost zahájení nové klasifikace s každým hodinovým cyklem. Kromě paměťové složitosti, hloubky řetězené linky a náročnosti na zdroje je proto dosažená pracovní frekvence vytvořené implementace jedním z důležitých a dále diskutovaných parametrů.

7.1 Paměťová složitost

První ze sledovaných vlastností u obou algoritmů byla paměťová složitost. Výsledky srovnání jsou vidět v grafu na obrázku 7.1. Paměťová složitost je zde vyjádřena jako počet uzlů konstruovaného rozhodovacího stromu. Na ose x jsou uvedeny jednotlivé sady pravidel, na ose y je potom znázorněn počet uzlů stromu s logaritmickou stupnicí. Barevně jsou v grafu uvedeny různé typy instrukcí. Oranžová barva znázorňuje instrukci využívající n-ární porovnání, zeleně je uvedena instrukce založená na principu hashování a červeně je znázorněna instrukce HyperSplit. Na první pohled je patrné, že instrukce HyperSplit vyžaduje ve všech případech výrazně větší množství paměti. Vlastní, nově navržené instrukce, které využívají oddělenou klasifikaci v jednotlivých dimenzích a zcela odlišný způsob konstrukce rozhodovacího stromu založený na zpětném navracení, přitom dosahují značné paměťové redukce. Měřítko na ose y je logaritmické a rozdíl hodnot paměťové složitosti mezi instrukcí HyperSplit a druhými dvěma instrukcemi je v některých případech i více jak dva řády.

Zaměříme-li se na výsledky paměťové složitosti u dvojice vlastních instrukcí (n-árního porovnání a hashování), je vidět, že obě instrukce jsou řádově srovnatelné. Instrukce založená na hashování přitom dosahuje ve všech případech lepších výsledků. Při porovnávání většího množství prefixů se totiž naplno projevují výhody instrukce. Instrukce umožňuje porovnání libovolného počtu prefixů vždy ve dvou krocích, zatímco u n-ární instrukce počet



Obrázek 7.2: *Hloubka řetězené linky (počet úrovní rozhodovacího stromu).*

potřebných kroků se zvyšujícím se počtem porovnávaných prefixů roste. S rostoucím počtem prefixů potom kromě počtu listových uzlů roste také počet vnitřních uzlů ve vyšších úrovních použitého n-árního vyhledávacího stromu. Konkrétní hodnoty provedené paměťové analýzy na zvolených sadách jsou uvedeny dále v souhrnné tabulce 7.1.

7.2 Hloubka řetězené linky

Druhou ze sledovaných vlastností byla výška rozhodovacího stromu a jí odpovídající hloubka řetězené linky. Získané výsledky zobrazuje graf na obrázku 7.2. Na ose x jsou opět uvedeny jednotlivé sady pravidel, na ose y je potom znázorněna hloubka řetězené linky. Barevně jsou opět uvedeny stejné typy instrukcí jako v předchozím grafu. Červeně je znázorněna instrukce HyperSplit, oranžová barva znázorňuje instrukci využívající n-ární porovnání a zeleně je uvedena instrukce založená na principu hashování.

Získané výsledky hloubky řetězené linky nejsou tak jednoznačné jako v případě předchozí paměťové složitosti. Na první pohled si lze v grafu všimnout výrazně horších výsledků u instrukce n-árního porovnání (oranžově) pro některé sady pravidel (*ipc1-5K*, *nb1*, *nb3*, *nb4* a *nb5*). Je to způsobeno vyšší četností zpětných návratů z důvodu většího množství prefixů odpovídajících klasifikovanému paketu u těchto sad. Zpětné navracení ale na druhou stranu přináší výraznou úsporu paměti. Horší výsledky u instrukce n-árního porovnání jsou dány také větším počtem kroků potřebným k porovnávání většího množství prefixů. Instrukce založená na principu hashování v těchto případech dosahuje daleko lepších výsledků, neboť k porovnání libovolného množství prefixů potřebuje vždy maximálně dva kroky. Nejlépe se pro tyto sady pravidel chová instrukce HyperSplit (červeně). Instrukce HyperSplit však nedosahuje nejlepších výsledků ve všech případech. Pro některé datové sady (*acl1* či *nb2*)

Tabulka 7.1: *Dosažená paměťová složitost (počet uzlů rozhodovacího stromu) a potřebná hloubka řetězené linky (počet stupňů rozhodovacího stromu) při použití různých instrukcí.*

Datová sada	Instrukce Hypersplit		Instrukce n-ární porovnání		Instrukce hashovací funkce	
	Uzly	Stupně	Uzly	Stupně	Uzly	Stupně
acl1	7 931	19	3 554	14	2 729	11
acl1-100	1 345	15	482	12	405	14
acl1-1K	10 999	22	4 269	23	2 992	16
acl1-5K	55 997	24	22 963	24	16 874	21
fw1	75 157	24	1 756	25	1 200	18
fw1-100	7 887	20	460	18	361	17
fw1-1K	468 379	28	11 424	40	7 739	31
fw1-5K	31 889 255	34	105 276	51	69 823	31
ipc1	381 069	26	17 162	46	11 652	34
ipc1-100	3 439	18	544	15	431	17
ipc1-1K	191 059	26	9 994	44	7 601	37
ipc1-5K	3 281 619	32	102 616	73	73 250	53
nb1-m05-m05	1 070 875	27	33 515	78	21 621	45
nb1-m05-05	1 449 169	27	37 914	77	24 287	46
nb1-05-m05	5 079 277	29	67 822	90	43 552	50
nb1-05-05	3 774 295	28	52 087	93	33 508	49
nb2-m05-m05	482 527	23	77 944	17	44 612	9
nb2-m05-05	470 823	24	77 493	20	44 342	9
nb2-05-m05-100	2 021	15	765	11	545	8
nb2-05-m05-250	5 865	16	1 942	16	1 382	9
nb2-05-m05-500	9 761	18	3 176	16	2 094	9
nb2-05-m05	648 421	23	69 175	20	39 610	9
nb2-05-05	734 015	24	65 813	19	37 794	9
nb3-05-05-500	165 047	23	12 082	69	8 021	40
nb4-05-05-500	301 989	24	43 598	90	26 659	52
nb5-05-05-500	433 807	24	19 430	90	12 900	51

dosahuje vlastní přístup (především instrukce založená na hashování) ještě lepších výsledků. Hloubka řetězené linky je pro tyto sady až o polovinu nižší než u instrukce HyperSplit. Sady jsou přitom charakteristické tím, že se pro ně zdrojový, respektive cílový port v pravidlech prakticky nevyskytuje. Přesné hodnoty výšky rozhodovacího stromu a odpovídající hloubky řetězené linky lze nalézt v souhrnné tabulce 7.1.

Tabulka 7.1 doplňuje veškeré výsledky znázorněné v grafech na obrázcích 7.1 a 7.2 o konkrétní hodnoty sledovaných parametrů. Řádky tabulky odpovídají jednotlivým datovým sadám s filtračními pravidly, ve sloupcích jsou pak uvedeny konkrétní počty uzlů a stupňů řetězené linky odpovídající jedné ze tří instrukcí a dané sadě.

Tabulka 7.2: *Zdroje FPGA potřebné pro implementaci operace LPM a dosažená pracovní frekvence pro různé velikosti domény hodnot a různou kapacitu prefixů.*

Doména [bity]	Kapacita	LUTs	FFs	BRAMs	Frekvence [MHz]
16	255	537	767	1	312,82
16	511	875	594	2	325,37
16	1 023	991	653	3	342,43
16	2 047	1 073	714	4	325,19
16	4 095	1 195	777	8	302,26
16	8 191	1 300	842	16	337,88
32	255	1 618	1 024	2	323,24
32	511	1 744	1 117	3	319,47
32	1 023	1 833	1 098	4	283,62
32	2 047	1 916	1 151	7	282,02
32	4 095	1 987	1 205	13	280,58
32	8 191	2 115	1 260	24	275,56
128	255	5 173	2 974	5	309,90
128	511	5 593	3 205	9	266,28
128	1 023	5 865	3 144	13	290,37
128	2 047	6 153	3 293	21	278,86
128	4 095	6 318	3 443	38	277,44
128	8 191	6 731	3 594	73	275,46

7.3 Prostorová složitost

Poslední sledovanou vlastností je prostorová složitost, jež je vyjádřena množstvím zabraných zdrojů na čipu FPGA. Pro syntézu byl použit nástroj Vivado 2016.1 a veškeré výsledky procesu syntézy v následujících tabulkách 7.2 až 7.6 jsou uvedeny pro čip Xilinx Virtex-7 XC7VH580T [24], který je osazen na cílové akcelerační kartě COMBO-100G.

Tabulka 7.2 zobrazuje zabrané zdroje FPGA potřebné pro implementaci předřazené operace LPM. Výsledky jsou zaznamenány pro různé velikosti domény hodnot (16, 32 a 128 bitů) a také pro různou kapacitu operace LPM (v rozsahu 2^8 až 2^{13} prefixů). Kapacitou operace LPM je zde myšlen minimální garantovaný počet prefixů, které lze uložit. Ve sloupcích tabulky jsou uvedeny konkrétní hodnoty zabraných zdrojů FPGA – vyhledávacích tabulek (LUT), registrů (FF) a blokových pamětí (BRAM). Se vzrůstající velikostí domény hodnot i maximální kapacity operace LPM narůstají také nároky na jednotlivé zdroje. V posledním sloupci tabulky je rovněž uvedena odhadovaná pracovní frekvence pro jednotlivé konfigurace. Ve všech případech jsou splněny minimální požadavky na dosažení potřebné propustnosti (frekvence $\geq 200\,\text{MHz}$).

V tabulce 7.3 jsou uvedeny zdroje FPGA potřebné pro implementaci jednoho stupně řetězené linky při použití různých instrukcí (procesních elementů). Jednou z instrukcí je pro srovnání s nově navrženým přístupem i instrukce HyperSplit. První sloupec výsledků (LUT) ukazuje náročnost na kombinační logiku. Vlastní nově navržené instrukce vychází v porovnání s HyperSplit lépe, neboť umožňují řešení problému nad mnohem menší doménou hodnot a díky redukci počtu bitů tak zjednodušují realizaci multiplexorů a komparátorů. V tabulce dále vidíme, že instrukce založená na hashování je náročnější na zdroje, především z důvodu realizace výpočtu hashovací funkce (při implementaci byla konkrétně

Tabulka 7.3: *Zdroje FPGA potřebné pro implementaci jednoho stupně řetězené linky a dosažená pracovní frekvence při použití různých instrukcí (procesních elementů).*

	LUTs	FFs	Frekvence [MHz]
Instrukce HyperSplit	120	356	323,73
Instrukce n-ární porovnání	99	240	277,47
Instrukce hashovací funkce	111	384	374,95

použita funkce CRC). Druhý sloupec (FF) udává potřebný počet registrů. Instrukce využívající n-ární porovnání vychází ve srovnání opět nejlépe. Vyšší nároky u instrukce s hashovací funkcí potom vyplývají z nutnosti uchovávat a také předávat určitou stavovou informaci mezi jednotlivými stupni linky. Poslední sloupec tabulky opět udává odhadovanou pracovní frekvenci. I zde je splněn požadavek na dosažení dostatečné propustnosti.

Tabulky 7.4, 7.5 a 7.6 ukazují souhrné výsledky syntézy pro jednotlivé typy instrukcí a vybrané sady pravidel. Výsledky syntézy odpovídají zabraným zdrojům FPGA pro celou řetězenou linku (včetně předřazené operace LPM). Parametry linky byly přitom pro každou sadu nastaveny na minimální požadované hodnoty tak, aby bylo možné klasifikaci paketů podle dané sady pravidel realizovat s co nejmenšími nároky na zdroje a zároveň byla dosažena minimální potřebná pracovní frekvence. Znamená to například nastavení odpovídajícího maximálního počtu prefixů v jednotlivých dimenzích nebo přesné délky pipeline.

Tabulka 7.4 uvádí celkově zabrané zdroje při použití instrukce HyperSplit, tabulka 7.5 při použití instrukce využívající n-ární porovnání a tabulka 7.6 ukazuje instrukci založenou na hashování. V prvním sloupci každé z tabulek jsou uvedeny použité datové sady. Další tři sloupce vyjadřují zabrané zdroje – LUT, registry (FF) a blokové paměti (BRAM). Kromě konkrétních číselných hodnot jsou výsledky zabraných zdrojů uvedeny také jako podíl zabrané kapacity použitého čipu.

Za pozornost stojí srovnání výsledků zabraných blokových pamětí (BRAM) u jednotlivých instrukcí. Instrukce HyperSplit pro některé větší sady (*ipc1-5K*, *nb1-05-m05* a *nb2-05-m05*) mnohonásobně překračuje kapacitu čipu, zatímco vlastní instrukce využívající n-ární porovnání vyžaduje pro stejné sady maximálně jednu čtvrtinu až jednu třetinu celkové kapacity blokových pamětí na čipu. Ještě lépe v tomto porovnání vychází instrukce založená na hashování, která přináší další úsporu 20-30 %. Z provedené analýzy také vyplývá, že nové přístupy sice vyžadují více stupňů řetězené linky, ale tato skutečnost se v požadavcích na zdroje výrazněji neprojevuje. Instrukce HyperSplit vykazuje celkově nižší nároky na LUT

Tabulka 7.4: *Souhrnné výsledky syntézy pro vybrané sady pravidel a řetězenou linku s instrukcí HyperSplit.*

Datová sada	LUTs	FFs	BRAMs	Frekvence [MHz]
acl1-100	6 251 (1,73 %)	7 239 (1,00 %)	0 (0,00 %)	239,75
acl1-1K	8 930 (2,46 %)	8 457 (1,17 %)	44 (4,68 %)	229,15
acl1-5K	9 417 (2,60 %)	10 127 (1,40 %)	120 (12,8 %)	239,01
ipc1-100	7 716 (2,13 %)	7 529 (1,04 %)	18 (1,91 %)	239,87
ipc1-1K	10 621 (2,93 %)	9 284 (1,28 %)	392 (41,7 %)	239,01
ipc1-5K	53 646 (14,8 %)	22 396 (3,09 %)	6 528 (694 %)	223,71
nb1-05-m05	80 639 (22,2 %)	24 254 (3,34 %)	9 284 (988 %)	233,43
nb2-05-m05	9 037 (2,49 %)	8 381 (1,16 %)	1 044 (111 %)	250,38

Tabulka 7.5: *Souhrnné výsledky syntézy pro vybrané sady pravidel a řetězenou linku s instrukcí využívající n-ární porovnání.*

Datová sada	LUTs	FFs	BRAMs	Frekvence [MHz]
acl1-100	7 143 (1,97 %)	6 305 (0,87 %)	0 (0,00 %)	279,10
acl1-1K	11 584 (3,19 %)	9 757 (1,34 %)	26 (2,77 %)	226,65
acl1-5K	10 046 (2,77 %)	9 961 (1,37 %)	89 (9,47 %)	226,45
ipc1-100	9 144 (2,52 %)	8 222 (1,13 %)	0 (0,00 %)	269,18
ipc1-1K	17 930 (4,94 %)	15 832 (2,18 %)	83 (8,83 %)	233,05
ipc1-5K	41 193 (11,4 %)	33 243 (4,58 %)	328 (34,9 %)	222,72
nb1-05-m05	33 311 (9,18 %)	35 412 (4,88 %)	235 (25,0 %)	237,42
nb2-05-m05	8 296 (2,23 %)	9 339 (1,29 %)	227 (24,1 %)	226,55

Tabulka 7.6: *Souhrnné výsledky syntézy pro vybrané sady pravidel a řetězenou linku s instrukcí využívající hashovací funkci.*

Datová sada	LUTs	FFs	BRAMs	Frekvence [MHz]
acl1-100	7 349 (2,06 %)	7 167 (0,99 %)	0 (0,00 %)	304,51
acl1-1K	7 647 (2,11 %)	7 885 (1,09 %)	32 (3,40 %)	249,69
acl1-5K	8 745 (2,41 %)	9 207 (1,27 %)	71 (7,55 %)	249,69
ipc1-100	9 467 (2,61 %)	8 529 (1,18 %)	0 (0,00 %)	306,37
ipc1-1K	12 865 (3,55 %)	13 179 (1,82 %)	69 (7,34 %)	249,69
ipc1-5K	34 107 (9 40 %)	29 275 (4,03 %)	262 (27,9 %)	243,19
nb1-05-m05	22 004 (6,07 %)	28 108 (3,87 %)	169 (17,9 %)	249,69
nb2-05-m05	3 997 (1,10 %)	5 494 (0,76 %)	165 (17,6 %)	250,38

pro menší datové sady. Při extrémních požadavcích na blokové paměti pro větší datové sady však u HyperSplit roste i náročnost na kombinační logiku vlivem související režie. Situace se obrací ve prospěch vlastních instrukcí.

Díky hluboce řetězenému zpracování se podařilo dosáhnout vysoké pracovní frekvence (více než 220 MHz), čímž jsou splněny veškeré požadavky pro zajištění plné propustnosti 100 Gb/s.

Kapitola 8

Závěr

Cílem diplomové práce byl návrh a implementace hardwarové architektury pro filtraci paketů, která bude dosahovat plné propustnosti 100 Gb/s a umožní tak nasazení ve vysokorychlostních počítačových sítích. Úkolem bylo provést návrh řešení pro akcelerační síťovou kartu s FPGA s cílem nalezení vhodného kompromisu mezi časovou a paměťovou složitostí algoritmu a s ohledem na možnost využití paralelního zpracování v FPGA. Tento vytyčený cíl byl splněn.

V rámci řešení práce jsem se nejprve seznámil s hardwarovou platformou COMBO, akcelerační síťovou kartou COMBO-100G a vývojovým prostředím NetCOPE. Následně jsem se zabýval typickými aplikacemi filtrace paketů v dnešních počítačových sítích a dále zaměřil svou pozornost na obecný problém klasifikace paketů včetně specifikace požadavků na klasifikační algoritmy. Cílem obecného problému řešení klasifikace paketů je nalezení vhodného algoritmu, který bude na jedné straně dostatečně obecný, rozšiřitelný a současně bude podporovat vyhledávání ve více dimenzích s co nejvyšší kapacitou klasifikačních pravidel. Na druhé straně je požadováno, aby byl algoritmus rychlý, efektivní a jeho časová a prostorová složitost umožňovala implementaci na dostupných hardwarových platformách. Teoretický rozbor problematiky filtrace paketů jsem uzavřel studiem současných známých a moderních metod provádění klasifikace paketů a zhodnocením jejich výhod a nevýhod vzhledem k ostatním přístupům. Veškeré získané poznatky z této oblasti jsou shrnuty v úvodních kapitolách práce.

Aby bylo možné navrhnout nový systém klasifikace paketů, bylo nutné provést důkladnou analýzu vlastností filtračních pravidel a dostupných datových sad. Důležitou vlastností vyplývající z analýzy je nízký počet unikátních hodnot v jednotlivých dimenzích datové sady a nízký počet prefixů, které mohou odpovídat klasifikovanému paketu. Počet takových prefixů je prakticky nezávislý na celkovém počtu pravidel dané sady. Následně byla provedena také podrobná analýza časové a paměťové složitosti vybraných klasifikačních algoritmů pro dostupné množiny pravidel. Nejlepších výsledků dosáhly algoritmy založené na rozhodovacích stromech. Tyto algoritmy vykazují vynikající poměr mezi časovou a paměťovou složitostí a rovněž je možné je velmi dobře mapovat do tzv. hlubokých pipeline. Výsledky analýzy jsou shrnuty v kapitole 7 a tvoří základ návrhu výsledné architektury.

Na základě výsledků provedené analýzy byla navržena a implementována hardwarová architektura pro filtraci paketů pro vysokorychlostní počítačové sítě. Nová architektura umožňuje efektivní uložení datové struktury reprezentující sadu filtračních pravidel a je založena na hluboce řetězeném zpracování, které umožňuje dosažení vysoké pracovní frekvence. Řetězená linka slouží k provádění jednoduchého programu složeného ze sekvence instrukcí. Vykonávaný program je zde určitou reprezentací rozhodovacího stromu, přičemž

požadované redukce paměťové složitosti je dosaženo volbou vhodných instrukcí. Jako součást vlastní architektury byly rovněž navrženy a vytvořeny dvě instrukce. Jedna z instrukcí pracuje na bázi n-árního porovnávání, druhá je založena na principu hashování. Nad rámec zadání práce byl dále navržen zcela nový způsob konstrukce rozhodovacího stromu využívající zpětného navracení. Současně byl vytvořen i systém mapování rozhodovacího stromu do datové struktury vhodné pro řetězenou linku.

V závěru diplomové práce byla provedena důkladná analýza dosažených parametrů navržené architektury a vytvořené implementace. Ověření a vyhodnocení vlastností navrženého algoritmu bylo provedeno na dostupných množinách filtračních pravidel. Hlavním ze sledovaných parametrů byla paměťová složitost pro jednotlivé datové sady. Dosažené výsledky vlastního přístupu byly dále srovnávány s výsledky algoritmu HyperSplit, který při předchozí analýze dosahoval nejlepších výsledků z hlediska poměru mezi časovou a paměťovou složitostí. Díky hluboce řetězenému zpracování a vysoce optimalizované architektuře pro technologii FPGA bylo dosaženo vysoké pracovní frekvence (více jak 220 MHz), čímž byly splněny veškeré požadavky pro zajištění plné propustnosti 100 Gb/s. Nově navržený přístup založený na konstrukci rozhodovacího stromu s backtrackingem dosahuje také významné redukce paměťové složitosti. Ve srovnání s algoritmem HyperSplit se podařilo při přibližně stejném množství zabraných hardwarových zdrojů na čipu zredukovat paměťové požadavky v průměru o 72 %, v nejlepším případě až o 98 % a umístit tak na čip sadu až pěti tisíc filtračních pravidel.

Navrženou architekturu je možné dále rozšiřovat a optimalizovat. Jednou z možností je například rozšíření současné architektury o podporu dalších instrukcí. Nově navržené instrukce je potom možné vhodně sdružovat a kombinovat za účelem dosažení ještě vyšší efektivity vnitřní reprezentace při ukládání filtračních pravidel. Dalšímu výzkumu v této oblasti se předpokládám budu věnovat v rámci mojí disertační práce. V současné době je plánována publikace výsledků práce na konferenci FPGA 2017 [6] v USA. Dosažené výsledky diplomové práce budou rovněž využity v rámci spolupráce se sdružením CESNET při řešení výzkumného projektu technologické agentury ČR – Technologie pro ochranu vysokorychlostních sítí (TH01010229).

Literatura

[1] ANT@FIT. *Systémy pro detekci nebezpečného provozu, technická zpráva* [online]. 2010. [cit. 2015-12-26]. Dostupné na URL: http://www.fit.vutbr.cz/~korenek/grants.php?file=/proj/442/Public/Reporty&id=442

[2] ANT@FIT. *Hardwarová akcelerace klasifikace paketů, technická zpráva* [online]. 2010. [cit. 2015-12-26]. Dostupné na URL: http://www.fit.vutbr.cz/~korenek/grants.php?file=/proj/442/Public/Reporty&id=442

[3] BLOOM, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*. 1970. Ročník 13, č. 7: s. 422–426. ISSN 0001-0782.

[4] *The Bro Network Security Monitor* [online]. 2016. [cit. 2016-01-09]. Dostupné na URL: https://www.bro.org/

[5] DHARMAPURIKAR, S., SONG, H., TURNER, J., aj. Fast Packet Classification Using Bloom Filters. *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS 2006. New York, NY, USA, 2006, s. 61–70. ISBN 1-59593-580-0.

[6] FPGA 2017. *25th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* [online]. 2016. [cit. 2016-05-19]. Dostupné na URL: http://www.isfpga.org/

[7] FRIEDL, Š., PUŠ, V., MATOUŠEK, J., aj. *CESNET Technical Report: Designing a Card for 100 Gb/s Network Monitoring* [online]. 2013. [cit. 2015-12-26]. Dostupné na URL: https://www.cesnet.cz/wp-content/uploads/2014/02/card.pdf

[8] GUPTA, P. *Algorithms for Routing Lookups and Packet Classification*. Disertační práce. Stanford University, 2000.

[9] LAKSHMAN, T. V., STILIADIS, D. High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM 1998. New York, USA, 1998, s. 203–214. ISBN 1-58113-003-1.

[10] LIBEROUTER / CESNET TMC group. *Technologies* [online]. [cit. 2016-01-10]. Dostupné na URL: https://www.liberouter.org/technologies/

[11] MARTÍNEK, T., KOŠEK, M. NetCOPE: Platform for Rapid Development of Network Applications. *Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, DDECS 2008. Bratislava, SK, 2008, s. 1–6. ISBN 978-1-4244-2277-7.

[12] MATOUŠEK, P. *Síťové aplikace a správa sítí*. Brno: VUTIUM, 2014. 396 s. ISBN 978-80-214-3766-1.

[13] *Netfilter – firewalling, NAT, and packet mangling for linux* [online]. 1999-2014. [cit. 2016-01-09]. Dostupné na URL: http://www.netfilter.org/

[14] POLČÁK, L., MARTÍNEK, T., HRANICKÝ, R., aj. *Zákonné odposlechy v moderních sítích: Shrnutí výsledků skupiny pro zákonné odposlechy a projektu Moderní prostředky pro boj s kybernetickou kriminalitou na Internetu nové generace, technická zpráva* [online]. 2014. [cit. 2015-12-26]. Dostupné na URL: http://www.fit.vutbr.cz/research/view_pub.php?id=10788

[15] PUŠ, V., KOŘENEK, J. Fast and Scalable Packet Classification Using Perfect Hash Functions. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA 2009. New York, USA, 2009, s. 229–236. ISBN 978-1-60558-410-2.

[16] PUŠ, V. *Packet Classification Algorithms*. Disertační práce. Brno: FIT VUT v Brně, 2012.

[17] PUŠ, V., TOBOLA, J., KOŠAŘ, V., aj. Netbench: Framework for Evaluation of Packet Processing Algorithms. *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS 2011. Los Alamitos, USA, 2011, s. 95–96. ISBN 978-0-7695-4521-9. Dostupné na URL: http://www.fit.vutbr.cz/netbench

[18] QI, Y., FONG, J., JIANG, W., aj. Multi-dimensional packet classification on FPGA: 100 Gbps and beyond. *Proceedings of the International Conference on Field-Programmable Technology*, FPT 2010. 2010, s. 241–248.

[19] SINGH, S., BABOESCU, F., VARGHESE, G., aj. Packet Classification Using Multidimensional Cutting. *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM 2003. New York, USA, 2003, s. 213–224. ISBN 1-58113-735-4.

[20] *Snort.Org* [online]. 2016. [cit. 2016-01-09]. Dostupné na URL: https://www.snort.org/

[21] SONG, H. *Evaluation of packet classification algorithm* [online]. 2007. [cit. 2015-11-09]. Dostupné na URL: http://www.arl.wustl.edu/~hs1/PClassEval.html

[22] TAYLOR, D. E., TURNER, J. S. Scalable packet classification using distributed crossproducing of field labels. *Proceedings of the 24th IEEE International Conference on Computer Communications*, INFOCOM 2005. Miami, USA, 2005, s. 269–280.

[23] TAYLOR, D. E., TURNER, J. S. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking*. 2007. Ročník 15, č. 3: s. 499–511. ISSN 1063-6692.

[24] XILINX. *DS180 – 7 Series FPGAs Overview, Product Specification* [online]. 2015. [cit. 2016-01-07]. Dostupné na URL: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[25] XILINX. *Virtex UtraSCALE+* [online]. 2016. [cit. 2016-04-09]. Dostupné na URL: http://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html

[26] YAXUAN, Q., LIANGHONG, X., BAOHUA, Y., aj. Packet Classification Algorithms: From Theory to Practice. *Proceedings of the 28th IEEE International Conference on Computer Communications*, INFOCOM 2009. Rio de Janeiro, Brazil, 2009, s. 648–656.

Příloha A

Obsah CD

```
/
├── experiments/
├── netbench/
├── source/
├── textdp/
├── bp-xkucer73.pdf
└── README.txt
```

Adresář `experiments/` obsahuje použité datové sady s filtračními pravidly, zdrojové kódy vytvořených Python skriptů, které byly použity pro analýzu filtračních pravidel, klasifikačních algoritmů i vlastní implementované architektury. Součástí jsou i veškeré naměřené výsledky a výsledky provedeného procesu syntézy.

Adresář `netbench/` obsahuje použitou verzi knihovny NetBench.

Adresář `source/` obsahuje zdrojové kódy VHDL implementace navržené architektury včetně dílčích podkomponent, vytvořeného simulačního prostředí a potřebných součástí pro opětovné provedení syntézy.

Adresář `textdp/` obsahuje zdrojové soubory textu diplomové práce pro její možnou úpravu a opětovné vysázení systémem LaTeX, včetně zdrojových souborů použitých obrázků a grafů vytvořených nástrojem `gnuplot`.

Soubor `dp-xkucer73.pdf` obsahuje vysázený text diplomové práce ve formátu PDF.

Soubor `README.txt` obsahuje informace o adresářové struktuře a obsahu přiloženého CD. Dále poskytuje podrobné pokyny pro umístění implementované architektury do adresářové struktury platformy NetCOPE a instrukce pro využití překladového systému platformy pro spuštění procesu syntézy a simulace.