VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HARDWAROVÁ AKCELERACE OPERACE HLEDÁNÍ NEJDELŠÍHO SPOLEČNÉHO PREFIXU

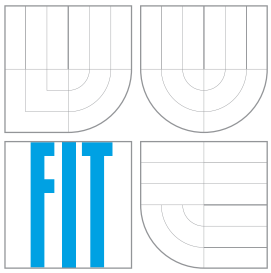
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ KEKELY

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HARDWAROVÁ AKCELERACE OPERACE HLEDÁNÍ NEJDELŠÍHO SPOLEČNÉHO PREFIXU

HARDWARE ACCELERATION OF LONGEST PREFIX MATCH

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ KEKELY

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KOŘENEK, Ph.D.

BRNO 2011

Abstrakt

V této bakalářské práci je popsán návrh a implementace hardwarové architektury na hledání nejdelšího shodného prefixu s ohledem na dosáhnutí rychlosti a propustnosti požadované v dnešních vysokorychlostních počítačových sítích. Zaměřuje se na IPv4 i IPv6 sítě. Navrhnutá hardwarová architektura dosahuje propustnost minimálně 75 Gbps na nejkratších IPv4 i IPv6 paketech. Výkonnost navrhnuté architektury je porovnána s výkonností zvolených, v současné době běžně používaných algoritmů. Jde konkrétně o algoritmy Tree Bitmap, Shape-Shifting Trie a Binary Search on Prefixes. Ty byly v rámci práce implementovány v jazyce C s využitím vícevláknového zpracování s ohledem na maximální využití výkonnosti dnešních vícejaderných procesorů.

Abstract

This bachelor's thesis describes design and implementation of hardware architecture for longest prefix match in order to achieve high throughput, which is required in today's high-speed computer networks. It is focused on IPv4 as-well-as IPv6 networks. Designed hardware architecture has throughput 75 Gbps on the shortest IPv4 and IPv6 packets. Performance of designed architecture is also compared with performance of chosen algorithms, which are used in nowadays commercial devices. These algorithms are: Tree Bitmap, Shape-Shifting Trie and Binary Search on Prefixes. All algorithms were implemented in C language using multi-threaded processing.

Klíčová slova

nejdelší shodný prefix, LPM, FPGA, hardware, vlákna, Trie, TreeBitmap, Shape-Shifting Trie, Binary Search on Prefixes, IP adresy

Keywords

longest prefix match, LPM, FPGA, hardware, threads, Trie, TreeBitmap, Shape-Shifting Trie, Binary Search on Prefixes, IP address

Citace

Lukáš Kekely: Hardwarová akcelerace operace hledání nejdelšího společného prefixu, bakalářská práce, Brno, FIT VUT v Brně, 2011

Hardwarová akcelerace operace hledání nejdelšího společného prefixu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jana Kořenka, Ph.D. Informace mi též poskytli lidé z projektu Liberouter a členové vývojové skupiny ANT.

.....
Lukáš Kekely
16. května 2011

Poděkování

Chtěl bych poděkovat hlavně vedoucímu práce Ing. Janovi Kořenkovi, Ph.D. za jeho odbornou pomoc a rady při psaní této práce. Chtěl bych též poděkovat lidem z projektu Liberouter a vývojové skupiny ANT za spolupráci a poskytnuté informace.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

Hlavným znakom dnešnej doby je rýchly rozvoj vo všetkých odvetviach ľudskej činnosti. Tento trend samozrejme neobchádza ani oblasť výpočtovej techniky. Práve naopak, táto je jednou z najrýchlejšie sa rozvíjajúcich oblastí. Jednou z najvýznamnejších oblastí v rámci výpočtovej techniky je oblasť sieťovej komunikácie a k nej neoddeliteľne patriaca celosvetová počítačová sieť (internet). Okrem lacnej a rýchlej komunikácie dnes internet poskytuje aj široký sortiment rôznych služieb. Od elektronického obchodovania, reklamy, cez zdieľanie multimédií, hranie hier až po spoznávanie nových ľudí na nedávno rozšírených sociálnych sieťach.

Význam internetu, aj napriek jeho terajšiemu obrovskému rozšíreniu, naďalej neúprosne rastie. Pribúda počet zariadení schopných komunikácie, počet užívateľov ako aj čas, ktorý každodenne trávia online. Rastie tiež množstvo samotných dát, ktoré medzi sebou užívatelia prenášajú. Toto všetko má za následok neustály nárast objemu dát tečúcich sieťou, ktorého hlavným dôsledkom je neustála potreba rýchlejšej a výkonnejšej infraštruktúry zabezpečujúcej prenos týchto dát.

Odpoveďou na neustále rastúce požiadavky na rýchlosť sieťových zariadení, môže byť využitie hardvérovo urýchľovaných riešení základných operácií sieťovej infraštruktúry. Softvérové riešenia pracujú všeobecne na univerzálnych výpočtových architektúrach. Oproti nim je hlavnou výhodou hardvérovo urýchľovaných technológií špecializácia a optimalizácia hardvérovej architektúry pre potreby riešenie konkrétnej úlohy. Špecializácia hardvéru prináša najmä možnosť využiť paralelné a zreťazené spracovania dát a možnosť využiť výpočtové jednotky optimalizované práve pre potreby danej úlohy. Výsledkom špecializácie je potom zefektívnenie využívania hardvérových zdrojov vedúce k poklesu ceny výsledného riešenia.

Základnými stavebnými kameňmi sieťovej infraštruktúry internetu sú zariadenia zvané smerovače. Ide o zariadenia zaručujúce elementárnu funkčnosť siete tým, že zabezpečujú správne nasmerovanie dát od ich pôvodcu až k stanovenému cieľu. Ak chceme teda významne zrýchliť infraštruktúru celej siete, musíme zrýchliť práve tieto zariadenia. A v nich predovšetkým operácie, ktoré sa využívajú pri smerovaní dát. Veľmi významnou z nich je operácia hľadania najdlhšieho zhodného prefixu, ktorou sa zaoberá táto práca.

Hľadanie najdlhšieho prefixu všeobecne pracuje s množinou rôzne dlhých prefixov, v ktorých je schopné pre každý vstupný reťazec, nájsť najdlhší zhodný z nich. V počítačových sieťach je každé zariadenie identifikované svojou adresou, unikátnym reťazcom. Zároveň sú zariadenia ležiace, z pohľadu siete, blízko seba združované do skupín a zdieľajú rovnaký prefix adresy. Dá sa to prirovnať ku poštovým adresám, kedy napríklad ľudia žijúci v jednom dome majú všetci rovnaký začiatok adresy a rozdiel je len v čísle vchodu a bytu. Smerovače

nepoznajú smery k jednotlivým zariadeniam, ale len k ich skupinám. Pri smerovaní dát teda, v snahe nájsť najlepšiu cestu, chcú vybrať čo najmenšiu skupinu zariadení, do ktorej patrí aj hľadaný cieľ. Čím menšia skupina, tým je prirodzene dlhší spoločný začiatok adries v nej. Teda najmenšia vyhovujúca skupina je definovaná najdlhším zhodným prefixom a ten je schopná nájsť práve spomínaná operácia hľadania najdlhšieho zhodného prefixu. Množinu prefixov potom tvoria adresy skupín zariadení v sieti a vstupný reťazec je adresa cieľu dát.

Cieľom tejto bakalárskej práce je vzájomné porovnanie výkonnosti dosiahnuteľnej softvérovým a hardvérovým riešením operácie hľadania najdlhšieho zhodného prefixu. V rámci softvérového riešenia sú vytvorené implementácií vybraných, dnes bežne používaných algoritmov hľadania najdlhšieho zhodného prefixu. Dôraz je kladený na dosiahnutie maximálneho výkonu s využitím výhod viacvláknového spracovania na moderných viacjadrových procesoroch. V rámci hardvérového riešenia ide o návrh a vytvorenie čo možno najefektívnejšieho hardvérového urýchlenia operácie hľadania najdlhšieho zhodného prefixu. Nad rámec zadania je výkonnosť implementovaných softvérových algoritmov otestovaná aj na embedded procesoroch bežne používaných v sieťových zariadeniach.

Táto práca je rozdelená do 6 kapitol. Kapitola 2 obsahuje popis základných princípov fungovania dnešných počítačových sietí, význam operácie hľadania najdlhšieho zhodného prefixu pri práci sieťových zariadení a nakoniec popis fungovania algoritmov Tree Bitmap, Shape-Shifting Trie a Binary Search on Prefixes. V kapitole 3 je popísaná implementácia vyššie uvedených algoritmov v jazyku C zameraná na maximalizovanie výkonnosti algoritmov s využitím výhod viacvláknového spracovania na moderných viacjadrových procesoroch. Kapitola 4 obsahuje popis návrhu a implementácie vlastnej hardvérovej architektúry vykonávajúcej operáciu hľadania najdlhšieho zhodného prefixu s využitím platformy FPGA. Kapitola 5 popisuje a porovnáva výsledky meraní výkonnosti navrhnutej hardvérovej architektúry a viacvláknových implementácií spomínaných algoritmov. Nakoniec v kapitole 6 je uvedené celkové zhrnutie obsahu a výsledkov práce.

Kapitola 2

Teoretický rozbor

V kapitole sú stručne zhrnuté všetky základné teoretické vedomosti, tvoriace základ praktickej časti práce popísanej v nasledujúcich kapitolách. Začiatok rozboru tvorí stručný popis základného modelu a princípu fungovania dnešných počítačových sietí. Potom nasleduje popis časti tohto modelu, ktorá je zameraná na doručovanie a riadenie pohybu dát sieťou. Hlavný dôraz je kladený na princípy adresovania zariadení a na prácu smerovačov pri výbere ciest pre dáta. Nakoniec nasleduje popis princípu fungovania konkrétnych bežne používaných algoritmov realizujúcich hľadanie najdlhšieho zhodného prefixu, ktoré boli využité pri implementácii softvérovej časti práce.

2.1 Princíp počítačových sietí

Text sekcie vychádza najmä z poznatkov uvedených v knihe [1]. Pri ľubovoľnom druhu komunikácie, teda aj sieťovej, musia byť prítomné prvky medzi, ktorými táto komunikácia prebieha. Minimálne prvok, ktorý vystupuje ako pôvodca správy a prvok, ktorému je správa určená, teda príjemca správy. V počítačových sieťach sa takéto zariadenia označujú ako koncové (ang. end devices) a medzi ne patria napríklad osobné počítače, IP telefóny alebo sieťové tlačiarne. S koncovými zariadeniami sa väčšina užívateľov bežne stretáva a sú im známe.

Samotný pôvodca a príjemca komunikácie nestačia, okrem nich musí vždy existovať ešte aj spôsob akým si budú navzájom predávať správy. V sieťach sa o predávanie správ pri komunikácii starajú zariadenia označované ako sprostredkovateľské (ang. intermediary devices). Sprostredkovateľské zariadenia sa starajú o bezporuchový prenos správ v sieti, ich správne smerovanie a čo najrýchlejšie doručenie na miesto určenia. Medzi tento druh zariadení patrí napríklad rozbočovač, smerovač alebo sieťový most. Väčšina bežných užívateľov sa so sprostredkovateľskými zariadeniami často nestretáva a väčšinou o ich prítomnosti a funkčnosti v sieti ani nevie.

Ďalším aspektom, ktorý podmieňuje schopnosť komunikácie je nutnosť mať isté prenosové médium schopné šíriť komunikáciu. V počítačových sieťach sú prenosovými médiami napríklad medené a optické káble alebo tiež bezdrôtové vysielanie (WiFi). Prenosovými médiami sú teda tvorené spoje medzi jednotlivými sieťovými zariadeniami, ktoré musia byť teda schopné zakódovať správy do formátu schopného prenosu po danom médiu.

Zabezpečením spomínaných troch aspektov komunikácie sme už schopní si vzájomne predávať správy a dáta. Ďalšie čo potrebujeme k dosiahnutiu plnohodnotnej komunikácie je systém riadenia komunikácie. V dnešných počítačových sieťach je základ riadenia založený

na modely TCP/IP, ktorým sú definované základné komunikačné protokoly, ktoré musia komunikujúce sieťové zariadenia dodržovať.

TCP/IP model vznikol už v 70. rokoch minulého storočia, vytvorila ho organizácia DARPA pôvodne pre armádu Spojených štátov [2]. Dnes je využívaný v drvivej väčšine sietí a je na ňom postavený aj základ dnešného internetu. Ide o protokolový model, ktorý obsahuje súbor implementácií základných služieb potrebných pre zabezpečenie úspešnej komunikácie a spojenie medzi zariadeniami v sieti. Ďalej definuje základné pravidlá pre formát, adresovanie a prenos dát.

Základným znakom TCP/IP modelu je rozdelenie zodpovednosti za jednotlivé operácie spojené s prenosom dát sieťou do abstraktných vrstiev (ang. layers). Každá z vrstiev má teda definovanú funkcionalitu a rozhranie, ktorým komunikuje s ostatnými vrstvami. Výhodou tohto prístupu je výrazné zjednodušenie vývoja a začlenenia nového protokolu, služby alebo zariadenia do existujúcej siete. Vďaka pevnému deleniu na vrstvy stačí implementovat len špecifickú funkcionalitu jednej vrstvy protokolu, na ktorej daná služba pracuje a ostatné vrstvy zachovať nezmenené. Táto vlastnosť sa dnes prejavuje napríklad fungovaním TCP/IP sietí nad rôznymi prenosovými médiami a to vždy len so zmenou najnižšej vrstvy.

Vrstvy v TCP/IP protokole sú celkovo štyri. Číslujú sa od jednotky pre najnižšiu vrstvu, teda vrstvu najbližšie k hardvéru a prenosovému médiu. Každá z vrstiev pridáva k zasielaným dátam vlastné riadiace informácie, takzvanú hlavičku a prípadne aj pätu. Tento proces sa nazýva zabaľovanie dát (ang. data encapsulation). Na strane odosielateľa je poradie vrstiev, pri zabalení dát, zhora dole. Na strane príjemcu, pri rozbalení dát, je to zdola hore. Model TCP/IP obsahuje konkrétne nasledujúce vrstvy:

Aplikačná vrstva je najvyššou vrstvou TCP/IP protokolu. Zabezpečuje predovšetkým reprezentáciu a zobrazenie dát užívateľovi. Taktiež zabezpečuje kontrolu a definuje pravidlá dialógu medzi komunikujúcimi koncovými zariadeniami. Na tejto vrstve pracujú protokoly jednotlivých sieťových aplikácií, ktoré väčšinou vnímajú nižšie vrstvy ako čiernu skrinku schopnú prenášať ich komunikáciu v počítačovej sieti. Väčšina z aplikácií komunikuje na princípe klient-server a medzi najznámejšie protokoly patria napríklad HTTP, FTP, Telnet, SSH ...

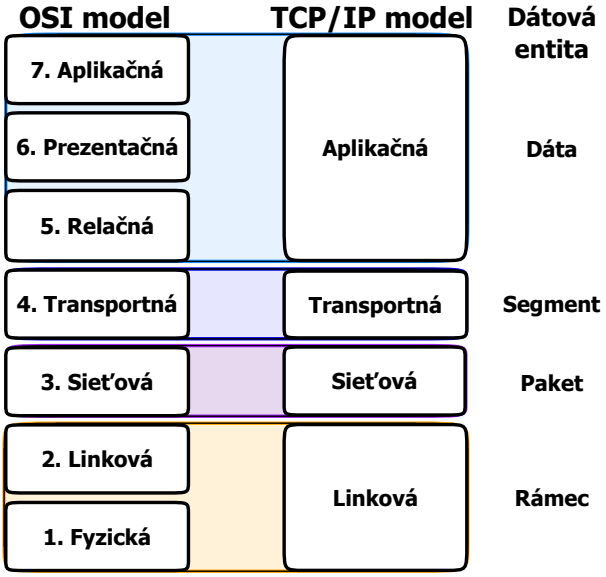
Transportná vrstva zabezpečuje spojenie medzi dvomi koncovými zariadeniami. Toto spojenie je nezávislé na použitých typoch sietí a prenosových médií. Definuje tiež mechanizmy kontroly zahltenia a toku, segmentáciu a kontrolu správnosti dát. Významná je aj možnosť adresovania konkrétnych aplikácií bežiacich na jednom koncovom zariadení, použitím čísel portov. Na transportnej vrstve pracujú protokoly TCP a UDP. TCP je protokol spájaný, ktorý poskytuje spoľahlivý prenos dát a zabezpečuje ich správnosť a integritu, nevýhodou však je relatívne vysoká réžia tohto prenosu. Naproti tomu protokol UDP je nespájaný, má nižšiu réžiu, avšak je nespoľahlivý, teda neposkytuje žiadne mechanizmy kontrolujúce správnosť prenosu dát. Prenášané dátové entity sú na transportnej vrstve nazývané segmenty (ang. segment).

Sieťová vrstva sa stará o zabezpečenie správneho doručovania segmentov a hľadanie najlepšej cesty pri doručovaní medzi sieťami. Tento proces sa označuje smerovanie a nie je spoľahlivý. Základnú funkčnosť vrstvy zabezpečuje protokol IP starajúci sa o adresovanie jednotlivých zariadení. Funkčnosť IP protokolu je podporovaná ďalšími protokolmi, medzi ktoré patria napríklad ICMP (správy o chybách a výnimočných stavoch), IGMP (správa multicastových skupín) a tiež smerovacie protokoly (EIGRP, OSPF, RIP). Prenášané dátové entity sú na sieťovej vrstve nazývané pakety (ang. packet).

Linková vrstva zabezpečuje ovládanie hardvérových zariadení a prenosových médií počítačových sietí. Popisuje postupy použité na kódovanie paketov na jednotlivých typoch prenosového média ako aj samotné pravidlá ich prenosu medzi zariadeniami pripojenými na rovnakej linke. Známe protokoly realizujúce linkovú vrstvu sú napríklad Ethernet a Token Ring. Prenášané dátové entity sú nazývané rámce (ang. frame).

Okrem modelu TCP/IP sa na popis fungovania počítačových sietí často používa aj model ISO/OSI, ktorý je rovnako ako TCP/IP vrstvový. Nie je však protokolovým modelom, ale referenčným. Jeho úlohou nie je špecifikácia implementácie a detailov jednotlivých vrstiev, ale skôr slúži ako pomoc na pochopenie funkcií a procesov využívaných v sieťovej komunikácii. ISO/OSI model pozostáva zo siedmych vrstiev, ktoré sú číslované rovnakým spôsobom ako pri TCP/IP.

Približné mapovanie funkčnosti jednotlivých vrstiev oboch modelov spolu s názvami používanými na označenie dátových entít na jednotlivých vrstvách je ukázané na obrázku 2.1. Najvyššia vrstva TCP/IP modelu spája funkčnosť troch najvyšších vrstiev ISO/OSI modelu. Vo väčšine sieťových aplikácií sa funkčnosť spomínaných troch vrstiev prelína a nie je možné určiť pevné hranice medzi nimi. Transportné a sieťové vrstvy sú v oboch modeloch funkčne ekvivalentné. Funkčnosť najnižšej vrstvy TCP/IP modelu spája zase funkčnosť najnižších dvoch vrstiev ISO/OSI modelu.



Obr. 2.1: Porovnanie vrstiev TCP/IP a ISO/OSI modelov

Ako už bolo spomínané, o smerovanie dát sieťou a ich prenos medzi koncovými zariadeniami, sa stará sieťová vrstva TCP/IP modelu a to bez ohľadu na protokoly vyššej vrstvy. Na zabezpečenie správneho fungovania tohto procesu je potrebné definovať niekoľko mechanizmov:

Adresovanie zariadení: každé z koncových zariadení musí mať priradenú unikátnu adresu, aby bolo možné jednoznačne rozlíšiť jednotlivé zariadenia

Zabaľovanie dát: kvôli potrebe nájsť správny cieľ kam konkrétny kus dát zaslať je aj ku dátam nutné pridať informácie o adresovaní, to je zabezpečené v rámci sieťovej hlavičky, ktorej súčasťou je okrem iného aj adresa príjemcu a pôvodcu paketu

Smerovanie: keďže komunikujúce koncové zariadenia nemusia byť vždy pripojené v rovnakej sieti, je nutné definovať spôsob akým budú pakety cestou riadené, aby správne dosiahli svoj cieľ, čo zabezpečujú sprostredkovateľské zariadenia (smerovače)

Rozbaľovanie dát: každé sieťové zariadenie musí byť schopné extrahovať a preskúmať cieľovú adresu prichádzajúcich paketov, ak sa navyše adresa zhoduje s jeho vlastnou, musí byť schopné z paketu správne rozbaliť v ňom zabalený segment a ten predať na spracovanie transportnej (vyššej) vrstve

Z pohľadu zamerania tejto práce sú detaily mechanizmov spojené so zabaľovaním a rozbaľovaním dát menej dôležité. Dôležité sú ale princípy spojené s adresovaním a smerovaním. Preto práve oni budú podrobnejšie rozobrané v nasledujúcej časti. Existuje viacero protokolov implementujúcich vlastnosti sieťovej vrstvy. Dnes najviac rozšírený a v internete bežne používaný je protokol IP (Internet Protocol). Ďalší popis je preto zameraný konkrétne na tento protokol.

Protokol IP bol definovaný v RFC 791 [3], dnes tvorí základný protokol používaný na doručovanie dát v internete. IP protokol poskytuje len základnú funkčnosť definovanú sieťovou vrstvou. Neobsahuje žiadne mechanizmy na sledovanie toku ani zaručenie spoľahlivého doručovania dát, to musia zariadiť protokoly vyšších vrstiev. Na druhej strane má však vďaka tomu protokol IP relatívne nízku réžiu prenosu.

Jedným z hlavných rysov IP protokolu je fakt, že pred začiatkom komunikácie nie je potrebné dopredu vytvárať žiadne spojenie. Príjemca dopredu nie je upozorňovaný na príchod paketov a tie cestujú sieťou každý individuálne. Zároveň dopredu ani nie je jasné, či vôbec daný príjemca existuje a či k nemu existuje cesta od odosielateľa. Preto môže dôjsť po ceste ku strate paketov alebo k ich príchodu v inom poradí, ako boli odoslané. Proces doručovania sa dá prirovnať k princípu akým funguje doručovanie zásielok poštou.

Ďalším rysom IP protokolu je doručovanie s najlepším úsilím (ang. best effort). Ide o transport paketov medzi zariadeniami so snahou vytvoriť čo najmenšiu záťaž na sieť aj za cenu občasnej straty paketu. Každé zariadenie sa však podľa svojich možností snaží o čo najlepšie zaslanie paketu smerom k jeho cieľu. Občasné straty paketov sú jedným z dôvodov nespoľahlivosti IP protokolu, nie všetky sieťové aplikácie však vyžadujú spoľahlivý prenos.

Posledným významným rysom IP protokolu je nezávislosť na prenosovom médiu. O túto závislosť sa z väčšej mieri starajú protokoly nižšej vrstvy. Z typu média však plynie jedno obmedzenie aj pre sieťovú vrstvu, ide o obmedzenie maximálnej veľkosti paketu. IP protokol na vyriešenie problému maximálnej veľkosti implementuje podporu pre rozdelenie (fragmentovanie) a následné znovu-poskladanie paketov pri prenose médiom.

V dnešnej dobe sa stretávame s dvomi bežne používanými verziami protokolu IP. Jedná sa o verzie IPv4 a IPv6 [4]. Rozšírenejšou je dnes ešte stále verzia 4, ale pomaly je nahradzovaná novšou verziou 6. Hlavným dôvodom prechodu na verziu 6 je postupné vyčerpanie adresného priestoru poskytovaného adresami IPv4. Dĺžka každej IPv4 adresy používanej na adresovanie jedného zariadenia je 32 bitov, maximálny teoretický počet adresovateľných zariadení je preto 2^{32}, čo je viac ako 4 miliardy adries. Tento počet však v dnešnej dobe už nestačí. Naproti tomu v IPv6 je dĺžka adresy 128 bitov a poskytuje teoretický počet zariadení rádovo 10^{38}.

Vo verzii 4 protokolu IP je 32 bitová adresa pri zápise rozdelená kvôli prehľadnosti na štyri 8-bitové čísla (ang. octets) oddelené bodkami, ktoré sa zapisujú v dekadickom tvare. Príkladom zápisu IPv4 adresy je napríklad adresa 192.168.1.2.

Vo verzii 6 je 128 bitová IPv6 adresa [5] pri zápise rozdelená na 8 skupín po 16 bitov. Skupiny sú oddelené dvojbodkami a sú zapisované ako hexadecimálne čísla. Príkladom zá-

pisu IPv6 adresy je napríklad adresa 2001:0DB8:0000:0000:0000:0000:1428:57AB. Pri IPv6 adresách je ešte možné využiť skrátený zápis, kedy je možné jednu ľubovoľnú neprerušenú postupnosť nulových skupín v rámci adresy vynechať. Vyššie spomenutú ilustračnú IPv6 adresu je možné zapísať aj v tvare 2001:0DB8::1428:57AB.

V počiatočných fázach sietí založených na IP protokole boli všetky zariadenia súčasťou jednej siete, neexistoval žiaden systém hierarchie adries ani delenia na celky. Ako však rástol počet pripojených zariadení, nastal problém s organizovanosťou adresovania a aj preťažovaním siete. Na odstránenie tohto problému dnes existuje mechanizmus delenia do podsietí (ang. subnets). Vďaka podsieťam je možné rozdeliť zariadenia do menších skupín, napríklad podľa vlastníka, polohy, účelu a zariadiť istú hierarchiu siete. Celej sieti potom stačí jediná (malý počet) prístupová brána (ang. gateway) na pripojenie k ostatným sieťam. Delenie na podsiete so sebou prináša výhody zvýšením výkonu, jednoduchším zabezpečením bezpečnosti a tiež jednoduchšou správou adries.

V protokole IP je delenie na podsiete riešené samotnými IP adresami. Každú IP adresu je možné logicky rozdeliť na dve časti. Prvá z nich identifikuje sieť, do ktorej dané zariadenie patrí a je označovaná za sieťovú časť. Druhá označuje konkrétne zariadenie v rámci jednej siete. Preto všetky zariadenia v rovnakej sieti zdieľajú spoločnú sieťovú časť IP adresy, inak povedané ich IP adresa má spoločný prefix.

Dátová šírka sieťovej časti IP adresy nie je pevne daná a je možné ju skoro ľubovoľne nastavovať. Zapísať dĺžku sieťovej adresy je možné dvomi spôsobmi, buď pomocou sieťovej masky alebo pomocou stanovenia dĺžky spoločného prefixu. V nasledujúcich odstavcoch sú uvedené príklady zápisu pre IPv4 adresu 192.168.1.1 a dĺžku jej sieťovej časti 24 bitov.

Sieťová maska má rovnakú dátovú šírku ako samotná IP adresa. Hodnota sieťovej masky v binárnom zápise začína postupnosťou jednotiek, za ktorou nasleduje postupnosť núl, pričom jednotky označujú bity patriace do sieťovej časti IP adresy a nuly označujú bity určujúce adresu zariadenia v rámci siete. Na určenie sieťovej časti IP adresy stačí použiť operáciu logického súčinu (logický *and*, $\wedge$) medzi IP adresou a príslušnou maskou. Maska sa zapisuje podobným zápisom ako IP adresa, teda napríklad môže byť adresa 192.168.1.1 a maska 255.255.255.0.

Adresa	**192** .	**168** .	**1** .	**1**
	1100 0000	**1010 1000**	**0000 0001**	**0000 0001**
		&		
Maska (/24)	**255** .	**255** .	**255** .	**0**
	1111 1111	**1111 1111**	**1111 1111**	**0000 0000**
		=		
Adresa siete	**1100 0000**	**1010 1000**	**0000 0001**	**0000 0000**
	192 .	**168** .	**1** .	**0**

Obr. 2.2: Výpočet sieťovej adresy pomocou masky

Proces aplikácie masky na IP adresu je ukázaný na príklade IPv4 adresy na obrázku 2.2. Najprv je IP adresa prevedená do binárneho zápisu. Potom je rovnako prevedená aj sieťová maska. Medzi vzniknutými binárnymi číslami je potom použitá spomenutá operácia logického súčinu a takto získaný výsledok je nakoniec prevedený z binárneho zápisu na štandardný zápis IP adries. Popísaným postupom je získaná adresa siete, do ktorej vzorová IP adresa patrí.

Zápis definovaním dĺžky spoločného prefixu je oproti maske jednoduchší, pre počítačové spracovanie je však zložitejšie potom určiť adresu siete danej IP adresy. Dĺžka spoločného prefixu sa zapisuje ako dekadické číslo za lomítkom nasledujúcim za normálnym zápisom samotnej IP adresy, napríklad 192.168.1.1/24. Pre IPv6 adresy sa väčšinou používa len takáto forma zápisu kvôli jednoduchosti.

2.2 Smerovanie

Text sekcie sa opiera o informácie uvedené v knihe [6]. Smerovaním (ang. routing) je v počítačových sieťach označovaný proces hľadania ciest pre dáta pri pohybe medzi zariadeniami. Smerovanie, ako už bolo spomínané, prebieha na sieťovej vrstve a jeho hlavnými vykonávateľmi sú sprostredkovateľské zariadenia, hlavne smerovače.

Smerovač (ang. router) je druh sieťového zariadenia zabezpečujúci prenos paketov medzi viacerými sieťami. Ide v podstate o zariadenie s účelom spájať počítačové siete na sieťovej vrstve TCP/IP protokolu. Každý smerovač obsahuje dve alebo viac sieťových rozhraní (ang. interface), z nich každé je zapojené a adresované v inej sieti. Podľa stavby je možné väčšinu smerovačov označiť za druh počítača, ktorý je v istých aspektoch fungovania podobný bežnému personálnemu počítaču. Okrem iného napríklad obsahuje základné komponenty ako procesor, pamäť RAM a ROM a vlastný operačný systém.

Pri určovaní výstupného rozhrania pre každý paket je snahou smerovača vybrať také, aby tento paket šiel čo najlepšou cestou, resp. aby dorazil čo najrýchlejšie k svojmu cieľu. Pri výbere cesty smerovač preskúma cieľovú IP adresu v sieťovej hlavičke paketu. Pre túto adresu potom vyhľadá najviac zhodnú sieťovú adresu zo svojej smerovacej tabuľky (ang. routing table). Súčasťou smerovacej tabuľky sú aj informácie o rozhraniach smerujúcich k daným sieťam. Každý smerovač vytvára svoje rozhodnutia pre smerovania paketov autonómne, len na základe vlastnej smerovacej tabuľky a bez priamej spolupráce s okolitými zariadeniami. Informácie od okolitých zariadení využíva smerovač len na doplnenie a úpravu informácií v smerovacej tabuľke.

Aby mohol pri smerovaní smerovač získať cieľovú IP adresu paketu, musí byť najprv schopný paket vybaliť z prijatého rámca linkovej vrstvy. Pred odoslaním na vybrané rozhranie ho musí zase správne zabaliť podľa typu linky pripojenej na toto rozhranie. Je veľmi časté, že paket je prijatý na inom type linky ako odoslaný. Smerovač musí byť preto schopný meniť typ sieťového rámca nesúceho paket pri jeho ceste medzi sieťami.

V dnešných sieťach sa na spájanie viacerých koncových zariadení používajú aj iné zariadenia ako len smerovače. Medzi najbežnejšie patria prepínače (ang. switch) alebo mosty (ang. bridge). Tieto zariadenia však vo všeobecnosti pracujú len na linkovej vrstve, preto poskytujú len fyzické spojenie medzi zariadeniami alebo sieťami. Nie sú schopné rozpoznávať rôzne logické siete a smerovať správne dáta medzi nimi.

Už spomínaná smerovacia tabuľka (ang. routing table) je základnou dátovou štruktúrou na uchovanie a reprezentáciu smerovacích informácií používaných smerovačom. Je uložená v jeho RAM pamäti a obsahuje informácie o priamo pripojených aj vzdialených sieťach. Na základe smerovacej tabuľky smerovač určuje akciu, ktorá sa vykoná s paketom resp. smer, ktorým bude tento paket odoslaný ďalej.

Informácie v smerovacej tabuľke sú ukladané tak, aby bolo možné čo najľahšie vykonávať vyššie spomenutú funkčnosť. Sú preto organizované do záznamov obsahujúcich asociácie medzi sieťovou adresou a adresou zariadenia (lokálnym rozhraním), ktoré vedie k danej sieti alebo je cieľom v nej. Záznamy v smerovacej tabuľke okrem toho obsahujú aj ďalšie informácie o ceste a ich základná logická štruktúra je nasledujúca:

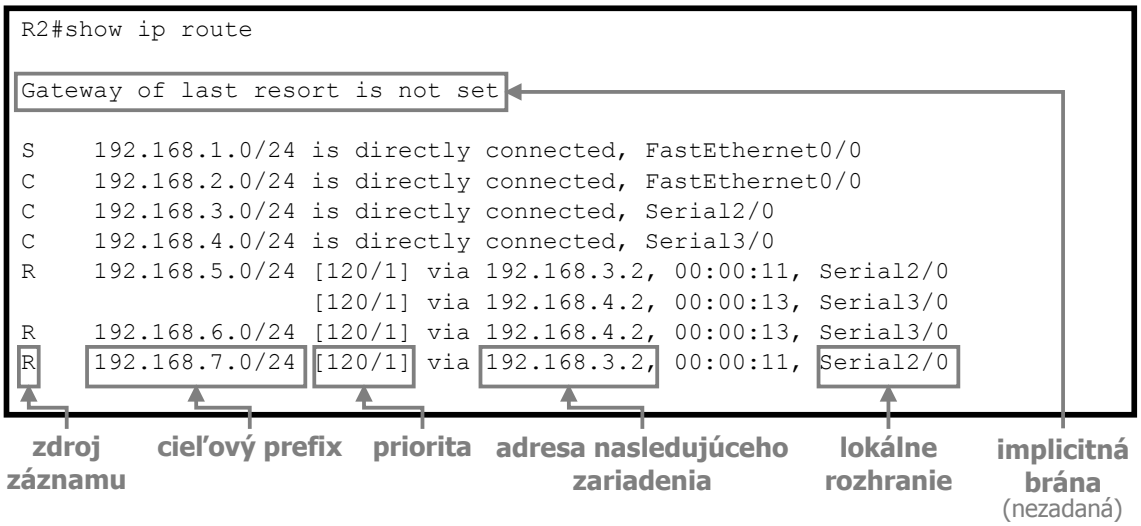
Zdroj informácie označuje zdroj smerovacej informácie v danom zázname. Existujú v zásade tri typy zdrojov. Prvým sú priamo pripojené siete, ktoré smerovač spozná samostatne pri nastavení a zapnutí rozhrania, na ktoré sú pripojené. Druhým sú statické cesty, ktoré sú zadávané ručne administrátorom pri nastavovaní smerovača. Posledným sú dynamicky zistené cesty, tie získava smerovač od okolitých zariadení za pomoci rôznych smerovacích protokolov.

Cieľová adresa je adresa, ktorej sa daný smerovací záznam týka. Cieľová adresa väčšinou nebýva adresou jedného zariadenia, ale celej siete. Ako už bolo spomínané skôr, zariadenia so spoločnou sieťovou adresou spolu zdieľajú rovnaký prefix IP adresy.

Priorita rozhoduje o výbere práve jedného záznamu pri existencii zhodných cieľových adries záznamov. V štandardnom nastavení platí, že lokálne pripojené siete majú prednosť pred statickými cestami a tie majú prednosť pred dynamickými cestami. Dynamické cesty majú priradené priority podľa smerovacieho protokolu, ktorým boli zistené a tiež podľa rôznych kritérií kvality danej cesty (napr. počet zariadení pred cieľom, rýchlosť linky).

Akcia určuje čo má smerovač spraviť s paketmi určenými pre adresy vyhovujúce danému prefixu. Ide v podstate o informáciu o lokálnom rozhraní alebo adrese zariadenia smerujúceho k danému cieľu. Táto informácia môže ukazovať priamo na sieť s adresátom paketu alebo len na susedné zariadenie, ktorým cesta k adresátovi vedie.

Okrem štandardných záznamov obsahuje smerovacia tabuľka aj jeden špeciálny záznam. Ide o adresu implicitnej brány (ang. default gateway). Na tú sú zasielané pakety v prípade, že ich cieľová adresa nezodpovedá žiadnemu z bežných záznamov. V prípade keď adresa implicitnej brány nie je nastavená, sú takéto pakety zahodené.



Obr. 2.3: Výpis obsahu smerovacej tabuľky na Cisco smerovači

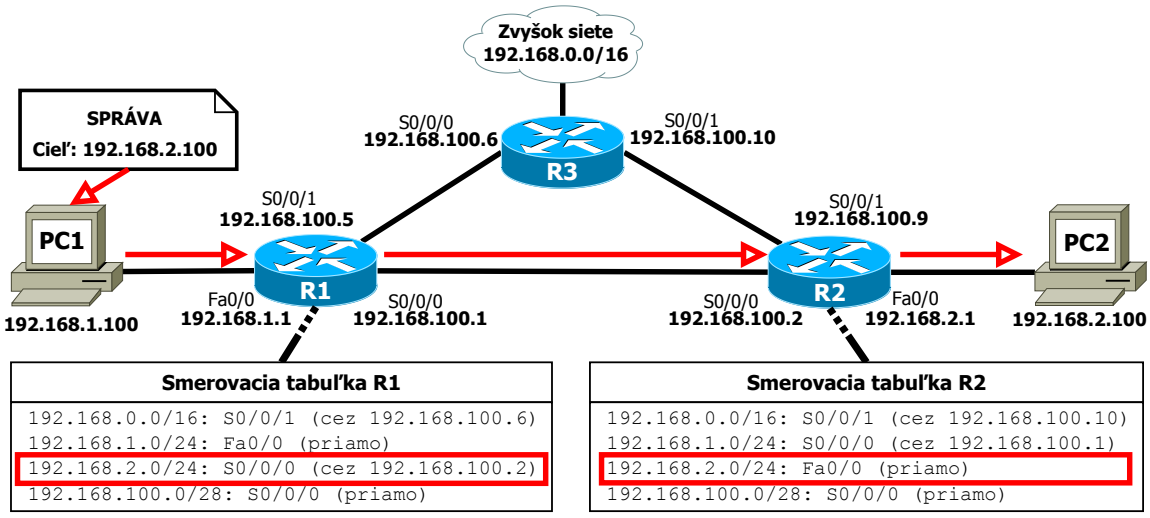
Príklad výpisu obsahu smerovacej tabuľke je možné vidieť na obrázku 2.3. Každý riadok reprezentuje jeden záznam a sivou farbou sú doplnené dodatočné vysvetlivky. V samotnej tabuľke je možné vidieť v prvých štyroch riadkoch informácie o štyroch lokálnych sieťach, tie sú priamo pripojené na uvedených rozhraniach smerovača. Ďalšie tri záznamy popisujú

vzdialené siete, pre ktoré musí byť okrem rozhrania uvedená vždy aj presná adresa nasledujúceho smerovača po ceste. V treťom zázname od konca je ukázaná situácia, kedy k jednej vzdialenej sieti vedú dve rovnako dobré cesty, smerovač potom môže zvoliť ľubovoľnú z nich.

Výber záznamu zo smerovacej tabuľky, ktorého akcia sa pri smerovaní použije, je vykonávaný pre každý prichádzajúci paket osobitne. Vždy je zvolený práve jeden záznam. Výber záznamu sa riadi podľa cieľovej adresy práve skúmaného paketu. V smerovacej tabuľke sa hľadá taký záznam, ktorý tejto adrese najlepšie zodpovedá.

Zodpovedá znamená, že adresa paketu má prefix zodpovedajúci cieľovej sieťovej adrese (prefixu) uvedenej v zázname smerovacej tabuľky. Napríklad IPv4 adresa 192.168.1.123 zodpovedá zo sieťových adries uvedených na obrázku 2.3 len adrese 192.168.1.0/24. Najlepšie v tomto význame prirodzene znamená, že z prefixov je vybraný vždy ten najdlhší zhodný. Napríklad, ak by v smerovacej tabuľke 2.3 bol naviac pridaný záznam so sieťovou adresou 192.168.0.0/16, IP adresa 192.168.1.123 by zodpovedala jemu a zároveň aj adrese 192.168.1.0/24. Uprednostnený by ale bol prefix 192.168.1.0/24 keďže je dlhším ($24 > 16$).

Príklad smerovania je možné vidieť na obrázku 2.4. Obrázok zobrazuje časť siete s adresou 192.168.0.0/16 a s centrálnym smerovačom R3. V sieti je naviac zapojená priama linka medzi R1 (sieť 192.168.1.0/24) a R2 (sieť 192.168.2.0/24) na odľahčenie toku cez R3. V označení rozhraní smerovačov sú použité skratky Fa pre FastEthernet a S pre Serial. Červenými šípkami je vyznačená trasa pohybu správy od PC1 ku PC2. Z PC1 je najprv zaslaná na smerovač R1. R1 sa rozhodne podľa svojej smerovacej tabuľky. Cieľová adresa správy 192.168.2.100 vyhovuje prefixom v 2 záznamoch, 192.168.0.0/16 a 192.168.2.0/24. Z nich je vybratý, 192.168.2.0 lebo je dlhší a správa je poslaná ďalej cez S0/0/0, využívajúc tak priamu linku medzi R1 a R2. Následne ju prijme R2, ktorý rozhodne podobne ako R1 len podľa svojej smerovacej tabuľky a vyberie záznam 192.168.2.0/24. Správa je tak zaslaná cez rozhranie Fa0/0 do cieľovej siete a doručená priamo PC2.



Obr. 2.4: Schéma smerovania správy sieťou

Celý popísaný výber záznamov pri smerovaní paketov je v podstate len operáciou hľadania najdlhšieho zhodného prefixu (ang. Longest Prefix Match alebo LPM). Pričom množina prefixov, nad ktorou prebieha toto hľadanie, je tvorená súborom prefixov sieťových adries zo záznamov smerovacej tabuľky. Vstupnou informáciou na vyhľadanie je cieľová IP adresa smerovaného paketu.

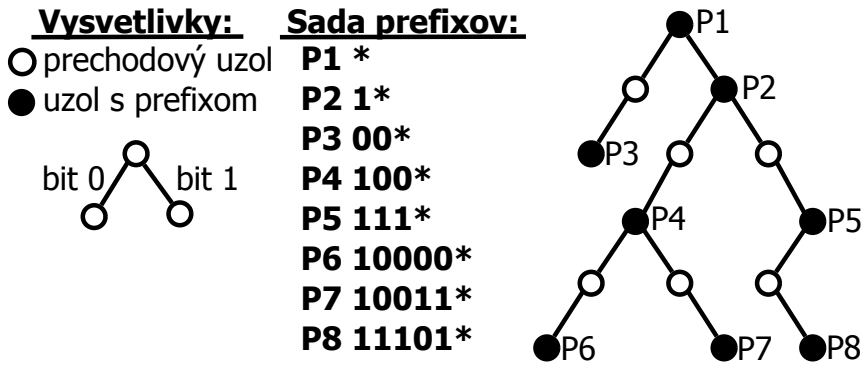
Algoritmov realizujúcich operáciu hľadania najdlhšieho zhodného prefixu existuje viacero. V práci sa zameriam len na vybrané 3 z nich. Konkrétne sú to algoritmy TreeBitmap, Shape-Shifting Trie a Binary Search on Prefixes. Prvé dva z uvedených algoritmov vychádzajú z konceptu vyhľadávania pomocou Trie a jeho všeobecného vylepšenia nazývaného rozšírený Trie (ang. Expanded Trie). Preto nasleduje najprv vysvetlenie konceptu Trie aj s popisom princípu spomenutého rozšírenia. Až za ním nasleduje popis samotných algoritmov. Posledný z uvedených algoritmov sa od predošlých principiálne líši, lebo nevyužíva Trie ale hašovacie tabuľky.

2.3 Trie

Trie [7], označovaný tiež prefixový strom, je usporiadaná stromová dátová štruktúra. V tej nie sú kľúče identifikujúce jednotlivé uzly ukladané priamo v uzloch, ale kľuč prislúchajúci danému uzlu je určený polohou tohto uzla v rámci stromu. Kľúčom je teda postupnosť hrán prejdených na ceste od koreňa stromu k danému uzlu. Hranám sú väčšinou priradzované jednoduché atomické hodnoty a kľúč sa tak zjednoduší na reťazec tvorený z týchto hodnôt. Obsah uzlov je tvorený už len samotnými dátami patriacimi ku konkrétnemu kľúču. Toto usporiadanie má za následok, že všetci potomkovia niektorého uzla s nim zdieľajú spoločný prefix kľúča a koreňový uzol má prázdny kľúč, teda prefix spoločný s každým uzlom stromu.

Algoritmy vyhľadania najdlhšieho zhodného prefixu využívajú verziu binárneho Trie, kedy každý uzol môže mať najviac dvoch priamych potomkov. Ďalej sa zavádza pevné značenie hrán: 0 pre hranu vedúcu k ľavému potomkovi a 1 k pravému. Kľúčom do binárneho Trie je potom binárna reprezentácia hľadanej IP adresy resp. prefixu sieťovej adresy. Dáta v jednotlivých uzloch sú potom len identifikátory záznamov smerovacej tabuľky patriace k príslušným prefixom.

Príklad Trie vytvoreného z jednoduchej sady prefixov je možné vidieť na obrázku 2.5. Vyplnené uzly predstavujú miesta ukončení prefixov s identifikátormi napísanými vedľa nich. Nevyplnené uzly sú len prechodové a sami nenesú žiaden prefix. Z vytvorenej Trie v obrázku je možné vidieť spomínané zdieľanie spoločnej časti prefixu potomkami rovnakého uzlu. Napríklad na prefixoch P4, P6 a P7. Uzly s prefixmi P6 a P7 sú oba nepriamymi potomkami uzlu s prefixom P4. Zároveň P6 a P7 oba začínajú trojicou 100, ktorá presne zodpovedá prefixu P4.

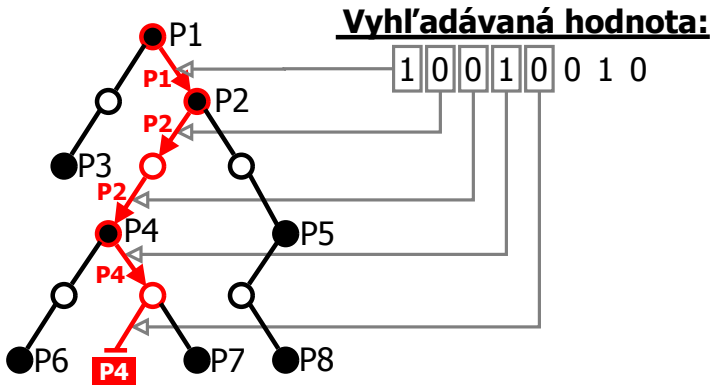


Obr. 2.5: Trie reprezentujúci jednoduchú prefixovú sadu

Vyhľadanie najdlhšieho zhodného prefixu v práve popísanej reprezentácii Trie začína vždy v koreňovom uzle. Z binárnej reprezentácie hľadanej IP adresy je v každom kroku

postupne vybraný práve jeden bit, počínajúc od najvýznamnejšieho (najviac vľavo). Tento bit priamo určuje hranu, ktorou sa vyhľadávanie posunie do ďalšieho uzla. Z ďalšieho uzla pokračuje vyhľadávanie rovnakým postupom, až kým nepríde do uzla, z ktorého nevedie hrana požadovaným smerom. Počas celej cesty je pre každý uzol overené, či neobsahuje platný identifikátor (teda či v ňom končí niektorý prefix). Počas prechodu stromom sa potom uchováva identifikátor z posledného takéhoto uzla, ktorý je po skončení prechodu aj výsledným identifikátor zodpovedajúci najdlhšiemu zhodnému prefixu vstupnej IP adresy.

Práve popísaný postup vyhľadávania je ilustrovaný na obrázku 2.6, pri určovaní najdlhšieho zhodného prefixu reťazca "10010010". Červenou farbou je označená prejdená cesta cez Trie. Sivou je označená závislosť každého výberu hrany na bite hľadanej hodnoty. Červené identifikátory prefixov napísané pri hranách označujú prefix, ktorý je v danom momente najdlhším nájdeným. V červenom obdĺžniku je potom uvedené celkové riešenie úlohy – najdlhší zhodný prefix nájdený prechodom Trie. Prechod začína v koreňovom uzle a postupuje smerom dole, vždy o jeden uzol v každom kroku, až kým nedôjde na miesto, z ktorého nevedie hrana požadovaným smerom.



Obr. 2.6: Prechod Trie pri vyhľadávaní najdlhšieho zhodného prefixu

Najväčšími výhodami Trie reprezentácie je jej jednoduchosť. Od nej sa potom odvíjajú aj časovo a algoritmicky nenáročné operácie pridávania a odoberanie prefixu z množiny. Takisto je možné dosiahnuť relatívne malé nároky na pamäť pre uloženie celého Trie. Veľkou nevýhodu je limitovanie rýchlosti vyhľadávania veľkým počtom prístupov do pamäte. Keďže je potrebné prechádzať strom po uzloch reprezentujúcich len jedno bitové úseky IP adresy. V najhoršom prípade je pri vyhľadaní potrebných až 32 náhodných prístupov do pamäte pre IPv4 adresu a až 128 pre IPv6.

Rozšírený Trie [8] pracuje s myšlienkou prechádzať v každom kroku hľadanú IP adresu po viacerých ako len jednom bite. Uzly základného Trie je kvôli tomu nutné združiť do takzvaných multi-uzlov, pre ktoré je nutné zvoliť vhodnú reprezentáciu. Rozšírený Trie prístup prináša výrazné zníženie počtu prístupov do pamäte a tým aj zrýchlenie celého vyhľadávania. Nevýhodou je väčšia zložitosť spracovania jedného uzla a tiež obtiažnejšie pridávanie a odoberanie prefixov. V niektorých prípadoch aj nárast pamäťovej náročnosti, vo všeobecnosti je však rýchlosť vyhľadávania kľúčová. Z myšlienky rozšíreného Trie vychádzajú aj nasledujúce dva popísané algoritmy.

2.4 Tree Bitmap algoritmus

Algoritmus Tree Bitmap (TBM) [9] nadväzuje na ideu rozšíreného Trie. Oproti iným podobným algoritmom ponúka rýchlejšie operácie pridania a odoberania prefixu ako aj nižšiu pamäťovú náročnosť. Pritom však zachováva všetky rýchlostné výhody spracovania vstupnej IP adresy po viacerých bitoch. Jeho základný koncept sa opiera o niekoľko poznatkov, z nich hlavné sú nasledujúce.

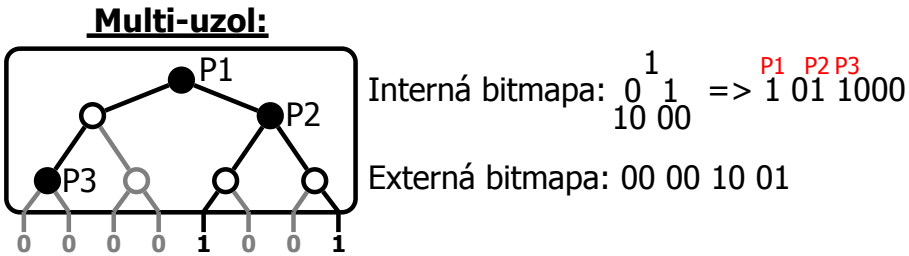
Prvým poznatkom je, že multi-uzol má v podstate dve základné funkcie, ktoré sú navzájom úplne nezávislé. Ide o nájdenie následníka aktuálne spracovávaného multi-uzla, v ktorom bude pokračovať vyhľadávanie a nájdenie identifikátora najdlhšieho zhodného prefixu v rámci spracovávaného multi-uzla.

Druhým poznatkom je schopnosť dnešných RAM pamätí, aj napriek pomalej frekvencii prístupov, získať z každého prístupu veľký počet dát. Dátová šírka štruktúry na uchovanie jedného multi-uzla môže byť preto relatívne veľká a je tak možné pokryť veľký počet uzlov Trie v jednom kroku. Zväčšovanie počtu uzlov v multi-uzle však nepriaznivo vplýva na čas potrebný na spracovanie takéhoto uzla. Preto ju nutné nájsť kompromis, ktorý bude vo výsledku najefektívnejší. Zároveň je výhodné zarovnávať veľkosti položiek dátovej štruktúry multi-uzla na hodnoty, ktoré sú mocninami 2. Keďže zarovnaný prístup do pamäte je efektívnejší a rýchlejší ako nezarovnaný.

Z uvedených poznatkov potom vychádzajú základné vlastnosti algoritmu. Prvou vlastnosťou je, že všetci následníci jedného multi-uzla môžu byť uložení za sebou. Potom stačí mať v každom multi-uzle len jeden ukazateľ na potomkov ukazujúci konkrétne na prvého z nich. Ukazovatele na ostatných potomkov je možné z neho odvodiť použitím offsetu. Podobný princíp môže byť použitý aj na identifikátory prefixov v rámci jedného uzla.

Ďalšou vlastnosťou algoritmu je, že každý multi-uzol je reprezentovaný s využitím len dvoch bitmáp. Jedna reprezentuje vnútorné uzly reprezentované v multi-uzle (interná) a druhá reprezentuje všetky existujúce hrany k následníkom daného uzla (externá). Obe bitmapy majú veľkosť 2^S bitov, kde S je hodnota stride popísaná ďalej.

Výhodou je pevný počet a tvar uzlov v podstrome, ktorý je reprezentovaný jedným multi-uzlom. Konkrétne ide o úplný binárny strom zvolenej výšky. Táto výška sa označuje termínom stride a je pre všetky multi-uzly stromu rovnaká. Pri skúmaní multi-uzla preto nie je potrebné dynamicky zisťovať jeho tvar a stačí používať len špecializovaný a hlavne optimalizovaný algoritmus výpočtu pre uvedený tvar.



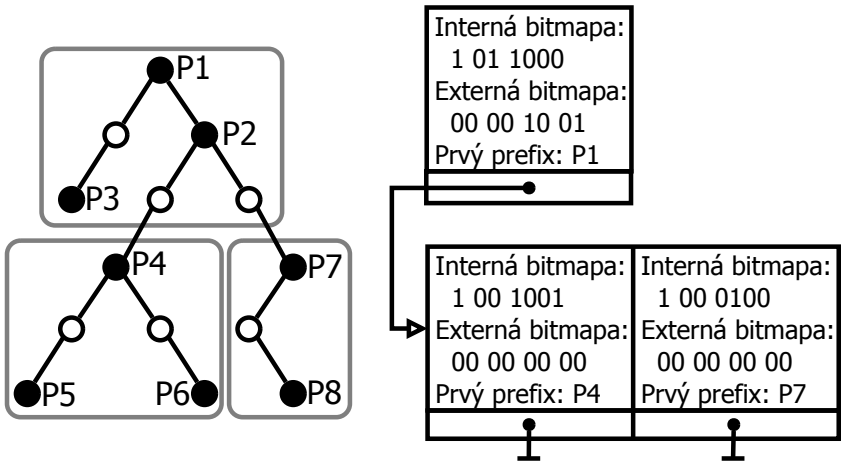
Obr. 2.7: Bitmapová reprezentácia multi-uzla v algoritme TBM

Princíp reprezentácie multi-uzla pomocou bitmáp je možné vidieť na obrázku 2.7. Hodnota stride je zvolená 3. Uzly Trie obsahujúce prefixy sú v internej bitmape na obrázku označené 1 a uzly neobsahujúce prefix 0. Interná bitmapa je potom vytvorená prechodom Trie v multi-uzli do šírky začínajúc od koreňa. Ako je možné vidieť, tento prechod udáva aj poradie v akom sú číslované prefixy v rámci multi-uzla. V externej bitmape sú 1 reprezen-

tované hrany k existujúcim následníkom a 0 k neexistujúcim. Externá bitmapa je vytvorená prechodom týchto hrán zľava doprava.

Reprezentovanie celého Trie pomocou Tree Bitmap je ukázané na obrázku 2.8. Zobrazená reprezentácia ukazuje využitie všetkých spomenutých kľúčových vlastností Tree Bitmap algoritmu pre stride s hodnotou 3. Každý sivý obdĺžnik v obrázku vymedzuje časť Trie reprezentovanú jedným multi-uzlom napravo. Multi-uzly napravo sú zároveň priestorovo rozložené rovnako ako časti ktoré reprezentujú. Všetky bitmapy sú vytvorené postupom popísaným v predošlom odstavci.

Vyhľadávanie v Tree Bitmap strome prebieha veľmi podobne ako vyhľadávanie v Trie. Začína sa v koreňovom uzle, výber následníka sa riadi bitmi hľadanej IP adresy a po ceste je vždy ukladaný aktuálny najdlhší prefix. Jediné čo stojí za zmienku, je spôsob akým sa z bitmáp vypočítava offset potrebný na určenie hodnoty konkrétneho následníka (identifikátora prefixu). Na výpočet je používaný algoritmus na spočítanie počtu jednotiek v príslušnej bitmape, ktoré ležia pred pozíciou určenou bitmi hľadanej IP adresy.



Obr. 2.8: Reprezentácia Trie pomocou TBM stromu

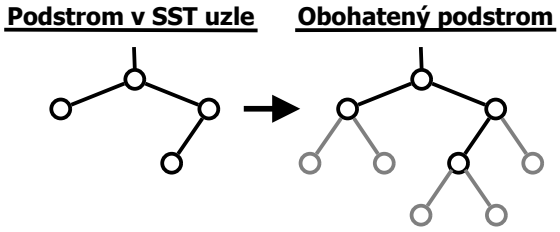
2.5 Shape-Shifting Trie algoritmus

Algoritmus Shape-Shifting Trie (SST) [10] nadväzuje na myšlienky rozšíreného Trie a vo svojej základnej podstate je veľmi podobný TBM algoritmu. Keďže TBM algoritmus má pevne dané jednotné tvary multi-uzlov, je jeho časová náročnosť lineárne závislá od dátovej dĺžky hľadanej hodnoty. Pre IPv6 adresy môže preto jeho výkonnosť výrazne klesať. Práve tento problém sa snaží vyriešiť SST algoritmus zavedením multi-uzlov s premenným tvarom, ktorý sa prispôsobuje konkrétnemu úseku stromu. Preto môže riedke stromy a stromy s dlhými nerozvetvenými úsekmi reprezentovať efektívnejšie. Inak algoritmus SST zachováva vlastnosti TBM algoritmu. Algoritmus TBM je v podstate len špeciálnou variantov algoritmu SST.

Multi-uzly SST algoritmu zodpovedajú istým podstromom originálneho Trie. Pričom existuje obmedzenie, že každý multi-uzol môže reprezentovať maximálne len K uzlov. Keďže ale nie je daný pevný tvar podstromov, je nutné tento tvar reprezentovať priamo v rámci multi-uzla. Na reprezentovanie tvaru slúži tvarová bitmapa (ang. shape bitmap, SBM), z ktorej každému uzlu postupne prislúchajú po 2 bity (jej veľkosť je maximálne 2K bitov).

Prvý z dvojice je pre ľavého potomka, druhý je pre pravého. Význam každého bitu z dvojice je určiť, či daný potomok patrí do rovnakého multi-uzla (1) alebo nie (0).

Pre ďalšie vysvetľovanie ešte definujme obohatenú verziu podstromu vnútri SST uzla. Prevod obyčajného podstromu na obohatený je ukázaný na obrázku 2.9. Obohatený podstrom je verzia, v ktorej sú do pôvodného postromu pridané pomocné uzly (sivé) tak, aby každý z plnohodnotných uzlov (čierne) mal priradených vždy práve 2 potomkov. Inak povedané, umelo pridané pomocné uzly sú na miestach, kde by mohli v pôvodnom Trie síce existovať uzly, ale už nie sú súčasťou tohto SST uzla (teda tam kde sú v tvarovej bitmape nuly).



Obr. 2.9: Prevod podstromu na obohatený podstrom

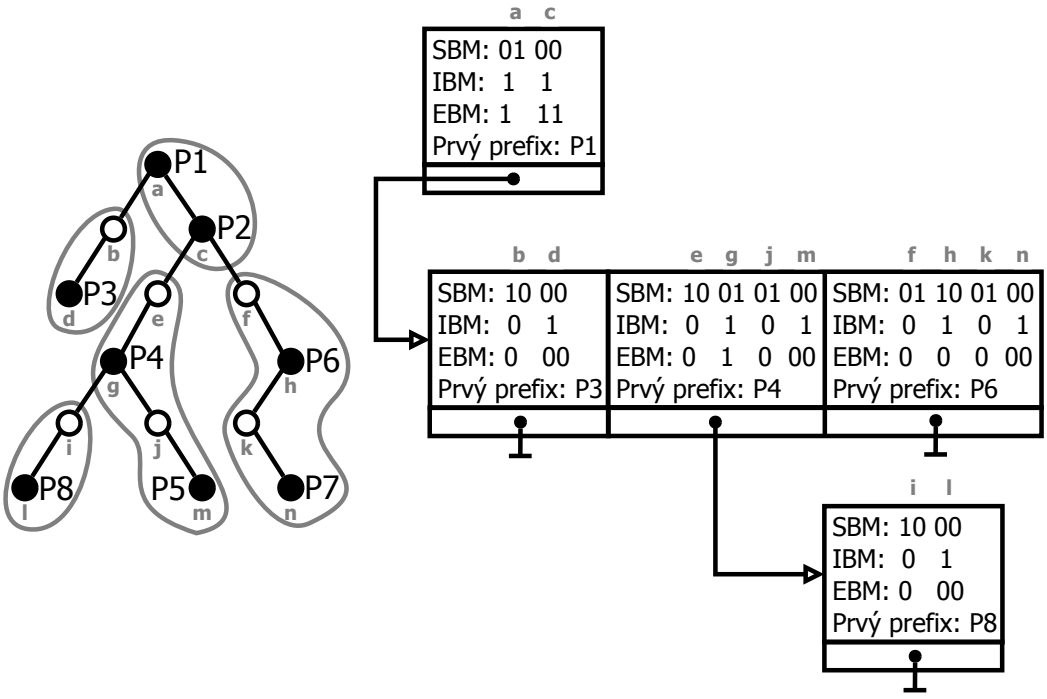
Okrem tvarovej bitmapy obsahuje každý SST uzol ešte významovo rovnaké bitmapy ako boli prítomné aj v TBM uzloch. Čiže internú (IBM, ukončenia prefixov vnútri uzla) a externú (EBM, existencia následníkov) bitmapu s dátovými šírkami K resp. K+1 bitov. Bity oboch bitmáp sú priraďované uzlom podľa ich poradia pri prechode do šírky podstromom SST uzla. Ďalej obsahuje SST uzol ešte aj (rovnako ako TBM uzol) ukazovateľ na začiatok poľa so svojimi následníkmi a identifikátor svojho prvého prefixu.

Príklad reprezentácie Trie pomocou algoritmu SST aj so zakódovaním obsahu jednotlivých SST uzlov je možné vidieť na obrázku 2.10. Sivé obdĺžniky reprezentujú jednotlivé SST uzly. Sivé písmená pri uzloch slúžia len ako ich pomocné pomenovania a nad bitmapami označujú uzol, ktorého sa bity v danom stĺpci týkajú. Na obrázku je možné vidieť napríklad spojitosť SBM a EBM, kedy ku každej nule v SBM prislúcha jeden bit v EBM. Taktiež každá dvojica bitov v SBM obsahuje vždy aspoň jednu jednotku okrem poslednej dvojice, ktorá je vždy 00 a funguje v podstate ako zarážka na zastavenie prechodu SST uzlom.

Vyhľadávanie v SST strome je v základnej podstate veľmi podobné Trie aj TBM algoritmu. Začína sa v koreňovom SST uzle, výber následníka sa riadi bitmi hľadanej IP adresy a po ceste je vždy ukladaný aktuálny najdlhší prefix. Keďže SST uzly môžu mať rôzny tvar, je spracovanie jedného multi-uzla náročnejšia úloha. Základom je dekódovanie tvarovej bitmapy. Hlavne je potrebné lokalizovať a spracovať tie bity, ktoré popisujú uzly ležiace na trase, ktorou prechádza vyhľadávanie príslušnej IP adresy.

Pri spracovaní prechodu jedným SST uzlom zavedieme nasledujúce premenné:

- $\mathbf{n}_i$ je počet uzlov vo vzdialenosti i od koreňa obohateného podstromu
- $\mathbf{f}_i$ je pozícia bitu v tvarovej bitmape, ktorý patrí prvému uzlu vo vzdialenosti i od koreňa
- $\mathbf{a}_i$ je i-ty bit IP adresy, ktorý je relevantný pre aktuálne spracovávaný SST uzol
- $\mathbf{p}_i$ je index v tvarovej bitmape, ktorý zodpovedá uzlu ležiacemu na ceste definovanej IP adresou vo vzdialenosti i od koreňa SST uzlu.



Obr. 2.10: Reprezentácia Trie pomocou SST stromu

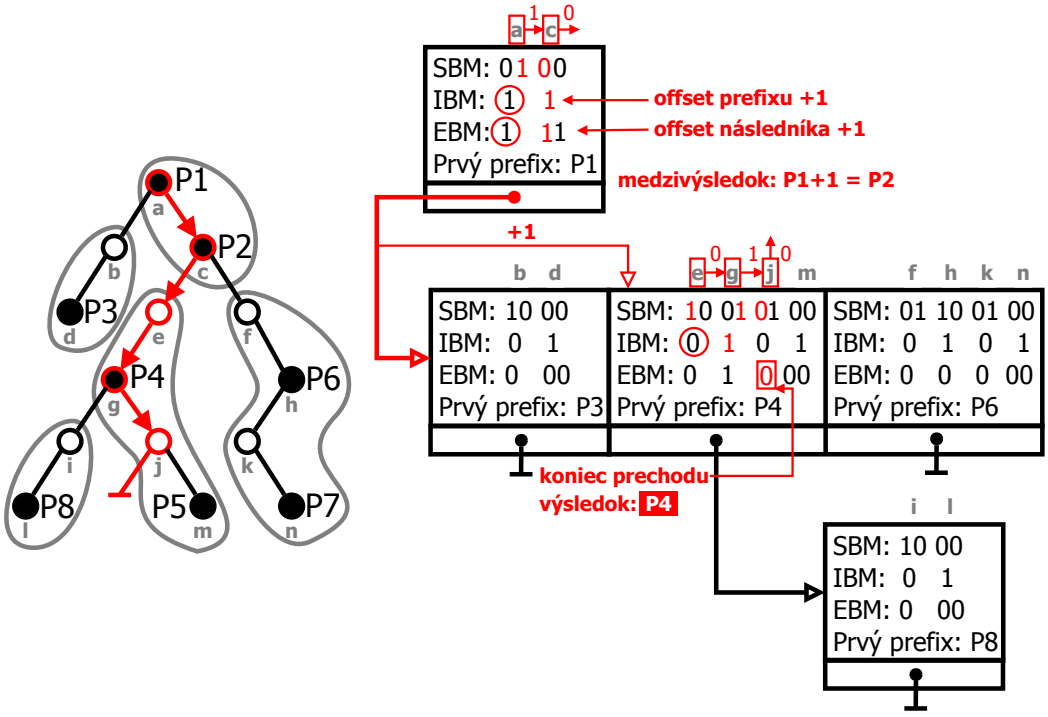
Okrem toho ešte definujme operáciu ones(i,j) ako počet jednotiek v tvarovej bitmape v rozmedzí medzi bitmi i až j. Pre vyššie definované premenné teraz platia vzťahy:

- $f_i = f_{i-1} + n_{i-1}$ kde $f_1 = 0$
- $n_i = 2 * ones(f_{i-1}, f_i - 1)$ kde $n_1 = 2$
- $p_i = f_i + 2 * ones(f_{i-1}, p_{i-1}) + a_i$ kde $p_1 = a_1$

Ak je i najmenším takým, že bit na pozícii p_i je nulový, potom p_i označuje pozíciu, kde vyhľadávací algoritmus opúšťa aktuálny SST uzol. Na určenie či a ktorým uzlom sa bude pokračovať, je potrebné použiť externú bitmapu. Index potrebného bitu v nej sa určí ako počet núl v tejto bitmape pred pozíciou $p_i - 1$. S internou bitmapou sa pracuje rovnako ako pri TBM algoritme. Takisto výpočet offsetov prebieha rovnako.

Príklad vyhľadávania je zobrazený na obrázku 2.11, pri určovaní najdlhšieho zhodného prefixu reťazca "10010010". Červenou farbou je vyznačený postup prechodu pri vyhľadaní hodnoty "10010010". Pre koreňový SST uzol sú hodnoty $n_1 = n_2 = 2$, $f_1 = 0$, $f_2 = 2$, $p_1 = 1$, $p_2 = 2$. Bit p_2 je prvým nulovým bitom v tvarovej bitmape, preto spočítame nuly naľavo od pozície 2 v SBM, ktorá je jedna. Vyberieme preto z EBM bit na pozícii 1, ktorý má hodnotu 1, teda vyhľadávanie pokračuje ďalším SST uzlom. V externej bitmape je pred pozíciou 1 práve jeden bit s hodnotou 1, preto bude offset do poľa následníkov 1. V ďalšom spracovávanom SST uzle sú hodnoty $n_1 = n_2 = n_3 = 2$, $f_1 = 0$, $f_2 = 2$, $f_3 = 4$, $p_1 = 0$, $p_2 = 3$, $p_3 = 4$. Bit p_3 je prvým nulovým bitom v tvarovej bitmape, preto spočítame nuly naľavo od pozície 4 v SBM, ktoré sú dve. Na pozícii 2 v EBM je bit s hodnotou 0 (neexistuje tade cesta ďalej), preto vyhľadávanie skončí. Výsledkom je posledný nájdený prefix, teda P4 nájdený v druhom kroku spracovania druhého SST uzla.

Vďaka variabilite tvaru SST uzlov je možné vytvoriť veľké množstvo reprezentácií toho istého Trie. Z pohľadu efektívneho spracovania je, ale logické požadovať optimálny tvar.



Obr. 2.11: Prechod SST stromom pri vyhľadávaní najdlhšieho zhodného prefixu

Existujú dva prístupy. Jeden prístup sa snaží minimalizovať celkový počet SST uzlov stromu a zároveň minimalizuje pamäťové nároky. Druhý sa snaží minimalizovať priemernú výšku platných ciest v strome a zároveň minimalizovať priemerný počet SST uzlov prejdených pri vyhľadaní IP adresy. Z pohľadu maximalizovania výkonnosti je výhodnejší druhý spomenutý variant.

2.6 Binary Search on Prefixes algoritmus

Algoritmus Binary Search on Prefixes (BSP, v preklade binárne vyhľadávanie na prefixoch) [11] sa od predošlých dvoch svojou podstatou výrazne líši. Základ vyhľadávania je postavený na metóde hašovania a využití hašovacích tabuliek.

Hašovacia tabuľka [12], tiež označovaná ako tabuľka s rozptýlenými položkami, je dátová štruktúra využívajúca istú hašovaciu funkciu na zabezpečenie mapovania medzi identifikátormi (kľúčmi) a k nim príslušnými dátami. Základom tabuľky je obyčajné pole prvkov adresované indexmi, v ktorom sú ukladané dvojice kľúč – dáta. Hlavnou úlohou hašovacej tabuľky je umožniť rýchly prístup k dátam podľa kľúča.

Hašovacia funkcia v tabuľke slúži na transformáciu hodnoty kľúča na hodnotu indexu poľa. Ideálne by mala každej hodnote kľúča priradiť unikátny index resp. unikátne miesto v poli. Vo väčšine prípadov tomu tak nie je, pretože pole v tabuľke je kvôli šetreniu pamäťou zámerne volené s menším rozsahom ako je rozsah hodnôt platných kľúčov. V tabuľke potom môže vznikať problém kolízií, kedy na jedno miesto v poli môžu byť mapované viaceré rôzne kľúče. Tento problém je v hašovacích tabuľkách riešený, ale má nepriaznivý vplyv na rýchlosť prístupu k dátam. Preto je snaha nastavovať parametre hašovacej tabuľky s ohľadom na minimilizovanie počtu kolízií. Dôležitými parametrami sú výber vhodnej hašovacej funkcie a veľkosti indexového poľa v tabuľke, vzhľadom na vlastnosti hodnôt kľúčov dát a ich počet.

V algoritme BSP sú hašovacie tabuľky využívané na reprezentovanie tried prefixov. Prefixy potom tvoria kľúče hašovacích tabuliek. Dáta sú identifikátory záznamov smerovacej tabuľky a pomocné informácie algoritmu patriace k daným prefixom (značky). Delenie do tried je na základe dĺžky prefixov, po jednej triede pre každú dĺžku vo vstupnom súbore. Zároveň je každá z tried reprezentovaná jednou samostatnou hašovacou tabuľkou. Pre zjednodušenie označovania je možné tabuľky a triedy označovať priamo dĺžkou prefixov, ktoré reprezentujú. Napríklad tabuľka dĺžky 8 je teda tabuľka reprezentujúca triedu s prefixmi dĺžky 8 bitov.

Aby nebolo nutné pri vyhľadávaní najdlhšieho zhodného prefixu pristupovať do každej z hašovacích tabuliek, je výhodné ich vhodne usporiadať. V BSP algoritme je usporiadanie riešené binárnym vyhľadávacím stromom. Hašovacie tabuľky tvoria uzly stromu a sú usporiadané podľa svojich dĺžok. Pre každý uzol stromu platí, že dĺžka v ňom uloženej tabuľky je väčšia ako každej v ľavom podstrome a zároveň menšia ako každej v pravom podstrome. Strom je ďalej zostavovaný, s ohľadom na efektivitu, ako výškovo vyvážený, teda s výškou logaritmicky závislou na počte uzlov.

Pri popísanom stromovom usporiadaní je nutné zaviesť mechanizmus rozhodovania o výbere následníka. Preto je nutné pridať do hašovacej tabuľky v každom uzle pomocné informácie. Ku každému prefixu pridáme značku o možnosti existencie dlhšieho zhodného prefixu pre hľadanú adresu. Značka teda konkrétne vraví o existencii prefixu v pravom podstrome uzla (dlhší prefix), ktorý má začiatok zhodný s daným prefixom. Ďalej je do hašovacej tabuľky uzla nutné, pre úplnosť značiek, umelo pridať nové prefixy. Vytvorené sú skrátením všetkých prefixov obsiahnutých v pravom podstrome, na dĺžku tabuľky. Ako dáta sú k umelo vytvoreným prefixom potom priradené rovnaké hodnoty aké patria ich najdlhším zhodným prefixom.

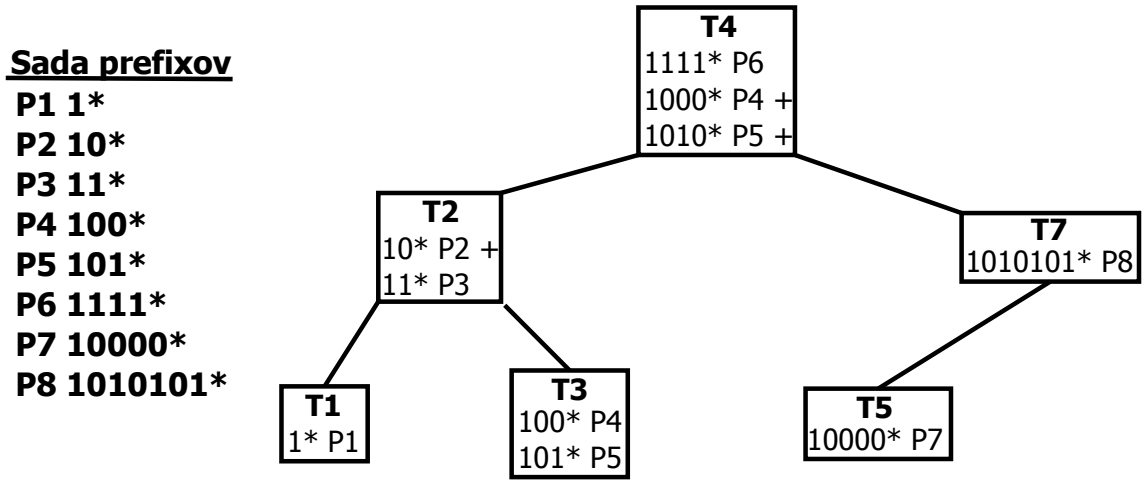
Príklad na vytvorenie stromu aj s obsahom prefixmi v hašovacích tabuľkách je možné vidieť na obrázku 2.12. Prefixy so značkou sú v obrázku označené znakom +. Tabuľky sú pomenované Tx, kde x reprezentuje dĺžku tabuľky (prefixov v nej) a nie je zobrazovaná ich skutočná podoba, len zoznam záznamov, ktoré obsahujú. Umelo pridané algoritmom boli v zobrazenom prípade dva prefixy v tabuľke T4, ide o 1000* a 1010*. Keďže ide o umelo pridané prefixy z dôvodu prefixov v tabuľkách T5 a T7, majú automaticky oba značku. Ku nim priradené identifikátory prefixov zodpovedajú ich najdlhšiemu zhodnému prefixu medzi kratšími prefixmi, teda 1000* má P4 od prefixu 100* a 1010* má P5 od 101*. Na obrázku je tiež možné vidieť, že prefix P2 má tiež priradenú značku. Je to kvôli prefixom P4 a P5, ktorých je prefixom. Pre P4 a P5 preto netreba umelo pridávať prefix do T2, stačí len P2 pridať značku.

Prechod stromom hašovacích tabuliek pri vyhľadávaní najdlhšieho zhodného prefixu je podobný prechodu Trie. Začína od koreňového uzlu a postupuje stromom po uzloch smerom nadol. V každom kroku prejde jeden uzol a rozhoduje sa o výbere následníka podľa informácie o type nájdeného zhodného prefixu v hašovacej tabuľke uzla. Môžu nastať tri situácie:

- **prefix so značkou** je uložený ako medzivýsledok a pokračuje sa pravým následníkom
- **prefix bez značky** je označený za celkový výsledok a vyhľadávanie končí
- **nenájdenie prefixu** znamená pokračovanie ľavým následníkom a zachovanie nezmeneného medzivýsledku

Popísaný princíp algoritmu BSP umožňuje vyhľadávanie s nižším počtom prístupov do pamäte. Počet krokov je vďaka usporiadaniu tabuliek do stromu zhodný s výškou stromu

a je teda logaritmicky závislý na dátovej šírke adresy. Nevýhodou však je možná nutnosť viacerých prístupov do pamäte na jeden výber z hašovacej tabuľky pri výskyte vyššieho počtu kolízií a tiež relatívne veľká pamäťová náročnosť.



Obr. 2.12: Vytvorenie BSP stromu z prefixovej sady

Kapitola 3

Viacvláknová implementácia

V kapitole je možné nájsť informácie o implementačných detailoch softvérových riešení algoritmov uvedených v predošlej kapitole. Algoritmy boli implementované presne podľa popisu so snahou o vytvorenie čo najefektívnejšieho zápisu. Hlavný dôraz bol kladený na operáciu vyhľadávania. Na vytváranie dátových štruktúr reprezentácie prefixových sád boli využité súčasti prostredia Netbench vyvinutého v rámci skupiny ANT. Implementácie využívajú viacvláknové spracovanie, preto sú v kapitole popísané tiež základné poznatky súvisiace s efektívnym návrhom viacvláknového spracovania úloh. Ako implementačný jazyk bol zvolený jazyk C. Jazyk C je imperatívny programovací jazyk pomerne nízkej úrovne. Medzi jeho hlavné výhody patria efektívnosť a automatické optimalizácie zdrojového kódu vedúce k vysokej výkonnosti výsledného programu. Význam pre viacvláknovú implementáciu má aj podpora správy vlákien pomocou štandardnej knižnice pthread.

Zdrojové texty softvérových algoritmov hľadania najdlhšieho zhodného prefixu boli implementované v rámci výskumnej skupiny ANT@FIT pôsobiacej na FIT VUT v Brne. Implementácie boli konkrétne vytvorené s využitím Netbench Frameworku, ktorého sú zároveň súčasťou. Netbench [13] je voľne dostupný framework implementovaný v jazyku Python, slúžiaci na podporu jednoduchého hodnotenia a experimentovania s rôznymi typmi algoritmov na spracovanie paketov. Cieľom je poskytnúť nezávislú platformu na porovnávanie základných parametrov algoritmov.

3.1 Viacvláknové spracovanie

Pri výkonnosti dosahovanej dnešnými procesormi je najslabším článkom, spomaľujúcim beh programu, rýchlosť prístupu do hlavnej pamäte počítača (RAM). Tá je štandardne rádovo pomalšia ako procesor. Keď program potrebuje dáta z pamäte, je nútený pozastaviť svoju činnosť a počkať na ich získanie. Čakanie na dáta plytvá procesorovým časom, lebo procesor ostáva nevyužitý. V dnešnej dobe sú bežne používanými viacjadrové architektúry. Ďalšia strata výkonu pramení z nevyužitia všetkých jadier procesoru pri nízkom počte vlákien, pretože jedno výpočtové vlákno je schopné bežať súčasne len na jednom jadre.

Oba spomínané problémy je možné riešiť využitím viacvláknovej implementácie. Výpočtový proces programu je vtedy rozdelený medzi viaceré vlákna bežiace paralelne. Každé vlákno teda môže bežať na jednom jadre a využiť tak celý procesor. Dnešné procesory navyše obsahujú mechanizmy na rýchle prepínanie behu vlákien. Keď niektoré vlákno dospeje do stavu, že musí čakať na dáta z pamäte, je jadro vláknu odobraté a pridelené inému vláknu, ktoré tak môže vykonávať užitočnú činnosť.

S možnosťou behu algoritmu na viacerých vláknach narastá schopnosť efektívne využiť procesor. Je ale potrebné určiť vhodný spôsob rozdelenia spracovania medzi jednotlivé vlákna. V zásade existujú dva základné prístupy delenia [14]. Prvá možnosť je rozdelenie algoritmu na výpočtovo rovnako náročné bloky, rozdeliť spracovanie blokov medzi vlákna a spojiť ich za seba (sériovo). Druhá možnosť je ponechať algoritmus nerozdelený a každému vláknu nechať spracovanie celého výpočtu potrebného pre jeden vstup, čo umožňuje súčasné paralelné spracovanie viacerých vstupov naraz.

Pri sériovom rozdelení nastáva problém ako vhodne rozdeliť a určiť časovo rovnako náročné časti algoritmu. Niektoré úlohy je dokonca nemožné rozdeliť na dostatočný počet častí alebo je procesný čas nestály. Pritom nevhodné rozdelenie vedie k zníženiu výkonnosti, pretože niektoré vlákna sú nútené čakať na dokončenie činnosti iných. Ďalším problémom sériového delenia je zabezpečenie komunikácie a správnej synchronizácie jednotlivých častí, tak aby nebola výrazne degradovaná výkonnosť algoritmu. K strate výkonnosti vedie aj nutnosť vykonania operácií spojených s prenosom dát medzi viacerými jadrami počas spracovania. Z uvedených skutočností jasne vyplýva, že výhodnejšie a jednoduchšie je paralelné spracovanie, pri ktorom každé vlákno implementuje celú funkčnosť algoritmu.

Pri paralelnom behu rovnocenných vlákien je potrebné vyriešiť spôsob rozdelenia vstupov medzi jednotlivé vlákna. Základný model s jednou vstupnou postupnosťou, z ktorej vlákna vyberajú striedavo, nie je efektívny. Synchronizácia výlučného prístupu k spoločnej postupnosti vstupov totiž prináša podobný problém ako pri sériovom spracovaní. Vtedy sú vlákna nútené čakať na dokončenie výberu iným vláknom. Operácie spojené so zabezpečením výlučného prístupu tiež pridávajú vlastnú réžiu. Výhodnejším modelom je preto vytvorenie nezávislej vstupnej postupnosti s výhradným prístupom pre každé vlákno.

Na prácu s vláknami bola pri implementácii využitá knižnica jazyka C s názvom pthread, definovaná štandardom POSIX. Táto knižnica poskytuje jednoduché štandardizované rozhranie prístupu k plnému využitiu hardvérovej podpory vlákien v procesoroch. Obsahuje základné dátové typy a volania potrebné na tvorbu, sledovanie, synchronizáciu a ovládanie behu paralelných vlákien, zároveň podporuje prenositeľnosť kódu.

3.2 Testovacie prostredie

Aby bolo možné zistiť výkonnosť jednotlivých algoritmov, vytvoril som univerzálne prostredie pre ich testovanie. Implementácia hlavnej riadiacej časti prostredia je spoločná pre všetky algoritmy a konfigurácie. Implementovaná je v rámci modulu main zloženého zo súborov main.c a main.h. Úlohou časti je správna príprava dát pre moduly algoritmov, správa a vytváranie vlákien, správne ukončenie programu a dealokácia pamäte. Keďže úlohou programu je meranie výkonnosti vyhľadávania, je súčasťou riadiacej časti meranie času behu algoritmov. Meraný je len čas potrebný na vyhľadávanie (bod 4. až 5.). Podrobnejšie je riadiaca časť popísaná nasledujúcim pseudokódom:

1. Načítanie binárneho súboru so vstupnými dátami (IP adresami) na vyhľadanie.
2. Načítanie konfiguračných dát daného algoritmu.
3. Rozdelenie vstupných dát na rovnako veľké časti určené pre jednotlivé vlákna.
4. Vytvorenie požadovaného počtu vlákien a štart vyhľadávania každým z nich.
5. Čakanie na dokončenie činnosti vlákien.
6. Upratanie a dealokovanie dát.

Textové súbory s konfiguráciami pre algoritmy, sú vytvárané za pomoci Netbench Frameworku. Vytvárajú sa z textových súborov obsahujúcich prefixové sady. Kvôli vytvoreniu vhodného formátu konfiguračných súborov bolo nutné obohatiť triedy jednotlivých algoritmov v Netbench o funkciu výpisu dátových entít. Podrobnosti sú uvedené v popise jednotlivých algoritmov v ďalších častiach kapitoly.

Každý algoritmus je implementovaný v rámci samostatného modulu. Moduly sú označované skratkami algoritmov zavedenými v predošlej kapitole (TBM, SST a BSP) a zdieľajú rovnakú podobu základného rozhrania, čo umožňuje rovnaké riadenie behu. Preto postačuje implementovať hlavnú riadiacu časť programu len raz a výber algoritmu zaistiť parametrickým prekladom a priradením vybraného modulu. Základné rozhranie slúžiace na ovládanie každého z modulov s algoritmami pozostáva z nasledujúcich troch funkcií:

load zabezpečuje načítanie konfigurácie algoritmu zo súboru. Súčasťou konfigurácie je algoritmom využívaná reprezentácia množiny prefixov zapísaná v pevne danej podobe v textovej forme. Funkcia pokrýva všetky operácie nutné na alokovanie pamäte, načítanie a usporiadanie konfigurácie. Teda celkové pripravenie algoritmu na vyhľadávanie.

lookup funkcia zabaľuje vyhľadávanie najdlhšieho zhodného prefixu nad vstupnými dátami pre daný algoritmus. Tvar jej vstupného parametru je definovaný štruktúrou arg, definovanou v súbore main.h. Tvar rozhrania funkcie je ale prispôsobený potrebám knižnice pthread, keďže funkcia priamo implementuje celú funkčnosť jedného výpočtového vlákna.

clean sa stará o korektné upratanie a dealokovanie všetkých dát modulu, vráti modul do východiskového stavu pred volaním funkcie load.

V zdrojových textoch modulov majú názvy týchto funkcií tvar ALG_funkcia, kde ALG je nahradené skratkou označenia modulu podľa algoritmu (napr. TBM_lookup).

3.3 Algoritmus Tree Bitmap

Implementovaný je v rámci modulu TBM, tvoreného súbormi TBM.c a TBM.h. Algoritmus TBM pracuje so stromovou štruktúrou zloženou z multi-uzlov. Multi-uzol je reprezentovaný štruktúrou TBM_node. Strom je reprezentovaný dátovým typom TBM_tree, ktorý je len ukazovateľom na štruktúru TBM_node. Strom je alokovaný vo forme indexového poľa.

Modul TBM podporuje celkovo štyri rôzne nastavenia Tree Bitmap algoritmu, nastaviteľná je hodnota stride. Podporované sú hodnoty 3, 4, 5 alebo 6. Stride ovplyvňuje dátové šírky každej z bitmáp v TBM uzle, pre podporované hodnoty sú šírky postupne 8, 16, 32 a 64 bitov. Podporované hodnoty stride boli vybrané s ohľadom na dátovú šírku bitmáp tak, aby dosiahnuté šírky presne zodpovedali šírkam štandardných celočíselných typov jazyka C. Požadovaná hodnota stride sa nastavuje ako parameter pri preklade modulu.

Konfiguračné dáta algoritmu s reprezentáciou sád prefixov sú vytvárané s využitím triedy TreeBitmap v rámci Netbench. V súbore TBM.py je implementované rozšírenie spomínanej triedy o funkciu display_data určenú na výpis reprezentácie TBM stromu vo vhodnej textovej forme. Formát výpisu začína riadkom s dvomi číslami – počet uzlov stromu a počet prefixov v strome. Za nimi nasledujú riadky uzlov so štvoricami čísel – identifikátor prvého prefixu, identifikátor prvého následníka, interná bitmapa a externá bitmapa. Jeden riadok reprezentuje jeden TBM uzol a riadky sú zoradené v poradí prechodu TBM stromu do šírky od koreňa.

3.4 Algoritmus Shape-Shifting Trie

Implementovaný je v rámci modulu SST, tvoreného súbormi SST.c a SST.h. Algoritmus SST pracuje so stromovou štruktúrou zloženou z multi-uzlov. Multi-uzol je reprezentovaný štruktúrou SST_node. Strom je reprezentovaný dátovým typom SST_tree, ktorý je len ukazovateľom na štruktúru SST_node. Strom je alokovaný vo forme indexového pola.

Modul SST podporuje celkovo štyri rôzne nastavenia Shape-Shifting Trie algoritmu, nastaviteľná je hodnota K. Podporované sú hodnoty 4, 7, 15 alebo 31. Hodnota K ovplyvňuje dátové šírky každej z bitmáp v SST uzle. Podporované hodnoty K boli vybrané s ohľadom na dátovú šírku bitmáp podobne ako pri TBM. Požadovaná hodnota K sa nastavuje ako parameter pri preklade modulu.

Konfiguračné dáta algoritmu s reprezentáciou sád prefixov sú vytvárané s využitím triedy SST v rámci Netbench. V súbore SST.py je implementované rozšírenie spomínanej triedy o funkciu display_data určenú na výpis reprezentácie SST stromu vo vhodnej textovej forme. Formát výpisu je podobným TBM algoritmu a začína riadkom s dvomi číslami – počet SST uzlov stromu a počet prefixov v strome. Za nimi nasledujú riadky uzlov s päticami čísel – identifikátor prvého prefixu, identifikátor prvého následníka, tvarová bitmapa, interná bitmapa a externá bitmapa. Jeden riadok reprezentuje práve jeden SST uzol a sú zoradené v poradí prechodu SST stromu do šírky od koreňa.

3.5 Algoritmus Binary Search on Prefixes

Implementovaný je v rámci modulu BSP, tvoreného súbormi BSP.c a BSP.h. Modul BSP využíva modul htable implementujúci hašovaciu tabuľku (obsahuje súbory htable.c, htable.h a hash_function.c). Algoritmus BSP pracuje so stromovou štruktúrou, ktorá v uzloch obsahuje hašovacie tabuľky. Uzly stromu sú reprezentované štruktúrou BSP_node obsahujúcou štruktúru htable_t reprezentujúcu hašovaciu tabuľku. Strom je reprezentovaný dátovým typom BSP_tree.

Modul podporuje celkovo dve rôzne nastavenia algoritmu BSP, je možné vybrať z dvoch rôznych hašovacích funkcií. Jedna z funkcií predstavuje plnohodnotnú hašovaciu funkciu (Jenkins Hash [15]), zabezpečuje rovnomernejšie rozloženie mapovania kľúčov do tabuľky, ale je výpočtovo náročnejšia. Druhá funkcia je len výber potrebného počtu najvyšších bitov hašovanej hodnoty, ktoré nie sú nijako upravované. Získanie výsledku je preto rýchle a v podstate bez výpočtu, ale s horším výsledným rozložením kľúčov. Veľkosť poľa je pre každú tabuľku volená samostatne, podľa počtu prefixov, ktoré v nej budú uložené. Veľkosť je vždy volená ako najnižšia mocnina dvojky vyššia ako ukladaný počet prefixov v tabuľke.

Konfiguračné dáta algoritmu s reprezentáciou sád prefixov sú vytvárané s využitím triedy BSP v rámci Netbench. V súbore BSP.py je implementované rozšírenie spomínanej triedy o funkciu display_data určenú na výpis reprezentácie dát algoritmu vo vhodnej textovej forme. Výpis začína riadkom so štyrmi číslami – počet uzlov binárneho stromu algoritmu, celkový počet prefixov v tabuľkách, koreňový uzol a výsledok pri nenájdení prefixu (prefix /0). Za nimi nasledujú postupne reprezentácie hašovacích tabuliek v uzloch stromu, zoradené podľa prechodu stromu do šírky od koreňa. Každá reprezentácia tabuľky začína riadkom so štyrmi číslami – dĺžka prefixov v tabuľke, ich počet, ľavý následník a pravý následník. Potom nasleduje zoznam prefixov v tabuľke aj s k nim patriacimi identifikátormi a značkami.

Kapitola 4

Hardvérová architektúra

Kapitola popisuje návrh a implementáciu algoritmu vyhľadania najdlhšieho zhodného prefixu vytvoreného s ohľadom na rýchlosť spracovania a priepustnosť jeho hardvérovej implementácie. Implementácia bola vytvorená na čipe technológie FPGA v rámci projektu NIFIC vývojovej aktivity Liberouter. Hlavné požiadavky kladené na architektúru boli:

- základná podpora IPv6 adries
- kapacita rádovo niekoľko tisíc pravidiel
- dosiahnutie priepustnosti potrebnej pre 10 Gbps siete
- podpora bezstratového prepnutia kontextu za behu

Navrhnutá hardvérová architektúra bola vytvorená s ohľadom na splnenie všetkých uvedených vlastnosti.

4.1 Rozbor súčasných algoritmov

Pri vytváraní návrhu algoritmu pre hardvérovú implementáciu boli ako základ použité myšlienky a koncepty algoritmov popísaných v kapitole 2. Koncepty boli upravené tak, aby v najväčšej miere využívali výhody plynúce z hardvérovej implementácie. Snahou bolo tiež odhaliť hlavné výkonnostné problémy a nevýhody jednotlivých algoritmov. Pri tvorení výsledného návrhu bola potom snaha vyhnúť sa odhaleným problémom a využiť len výhody konceptu algoritmu.

Najvýznamnejšou vlastnosť ovplyvňujúca hardvérový návrh je schopnosť hardvéru vykonávať veľký počet paralelných operácií súčasne. Využitie popísanej vlastnosti je v možné algoritme BSP. V BSP potom nie je potrebné hašovacie tabuľky usporadúvať do binárneho stromu, v ktorom je nutné pristupovať postupne do viacerých tabuliek sekvenčne za sebou. V hardvérovej architektúre je možné pristúpiť v jednom kroku do všetkých tabuliek súčastne, v ďalšom kroku potom už len stačí vybrať najdlhší z výsledkov získaných z tabuliek a získame celkové riešenie. Popísané usporiadanie oproti BSP zároveň ruší nutnosť existencie a ukladania značiek dlhších prefixov, čím čiastočne znižuje pamäťovú náročnosť algoritmu. Popísaný koncept bol predstavený aj v návrhu algoritmu Multimatch [16]. Zavedením popísaného konceptu sa zníži časová náročnosť algoritmu na konštantnú (O(1)). Významnou nevýhodou však stále zostáva veľká pamäťová náročnosť, ktorá ešte vzrastie pre IPv6 adresy. Problémom je tiež nutnosť podpory dynamického delenia pamäte medzi triedy prefixov, ktorá je v hardvérovom návrhu ťažko dosiahnuteľná.

Stromové algoritmy vychádzajúce z rozšíreného Trie (TBM a SST) majú hlavnú nevýhodu v sekvenčnom prechode poschodí stromu. Časová náročnosť (počet krokov prechodu) je potom lineárne závislá na dĺžke IP adresy. Použitie stromových algoritmov na 4-krát dlhšie IPv6 adresy spôsobuje 4-násobný pokles priepustnosti oproti IPv4 adresám. Výhodou sú však nižšie pamäťové nároky oproti hašovacím tabuľkám. Stromové algoritmy je teda vhodné používať pri vyhľadávaní zhodných prefixov pre hodnoty menších dátových šírok. Vtedy je počet sekvenčných krokov nízky a stratu výkonu na sekvenčnom spracovaní je možné kompenzovať použitím viacerých výpočtových jednotiek paralelne. Dôležitou podmienkou dostatočnej priepustnosti hardvérovej implementácie je nutnosť implementovať spracovanie jedného multi-uzla v čo najnižšom a hlavne konštantnom počte taktov. Z tohto pohľadu je algoritmus SST nevýhodnejší, pretože spracovanie bitmáp multi-uzla je ovplyvňované tvarovou bitmapou. Náročnejšie spracovanie SST uzla zároveň zvyšuje nároky na plochu čipu. Pre algoritmus TBM je možné implementovať spracovanie bitmáp multi-uzla v jednom takte.

4.2 Algoritmus Hash Tree Bitmap

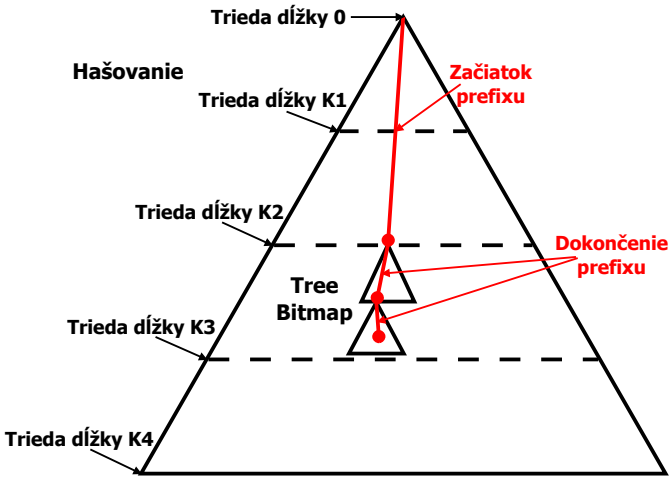
Algoritmus vychádza z vlastností popísaných v predošlej sekcii. Spája v sebe výhody hašovaného vyhľadávania spolu s výhodami stromového algoritmu Tree Bitmap. Je vytvorený špeciálne pre hardvérové použitie a zo softvérového pohľadu implementácie nie je efektívny.

Ako už bolo spomenuté v predošlej časti, pri delení prefixov do hašovacích tabuliek podľa dĺžky je nutné vhodne vyriešiť dynamické delenie pamäte. Nevýhodou pri delení prefixových sád do tried podľa dĺžky je nerovnomernosť veľkosti tried. Jednak neexistuje spôsob akým dopredu určiť, ktorá dĺžka prefixov bude ako zastúpená v prefixovej sade danej veľkosti. Zároveň niektoré dĺžky majú podstatne nižšie zastúpenie ako iné. Pre každú triedu je preto nutné použiť inak veľkú tabuľku a dopredu nie je možné určiť ako veľkú. V algoritme je preto zavedený koncept atomických tabuliek. Pamäť architektúry je rozdelená medzi dopredu pevne daný počet atomických tabuliek daných veľkostí. Každú atomickú tabuľku je možné nastaviť na ľubovoľnú platnú dĺžku prefixu. Atomická tabuľka neobsahuje mechanizmus na riešenie kolízie hašovacej funkcie. Kolízie sú riešené tak, že jednej triede prefixov je pridelených viac tabuliek a každá z kolidujúcich hodnôt je tak uložená v inej tabuľke.

Po zavedení konceptu atomických tabuliek má hardvérová architektúra dopredu daný počet atomických tabuliek. Aby nedochádzalo k neefektívnemu plytvaniu tabuľkami je nutné deliť prefixy na menej väčších tried. Popísané vlastnosti delenia sú dosiahnuté delením prefixov na triedy s rozsahom dĺžok. Pri pôvodnom delení by napríklad stačilo ak by v sade existoval jediný prefix dĺžky 9 a bolo by nutné kvôli nemu vytvoriť novú triedu a prideliť jej celú tabuľku. V novo zavedenom delení patria napríklad prefixy s dĺžkami 8 až 17 do jednej triedy a preto pri existencii jediného prefixu dĺžky 9 je tento prefix len pridaný do spoločnej triedy s ostatnými.

Pri delení prefixov do tried so súborom rôznych dĺžok nastáva problém s porovnaním vstupnej hodnoty s prefixmi rôznych dĺžok v rámci jednej triedy. Na riešenie tohto problému je využívaný algoritmus TBM. Je zavedené delenie prefixov na dve časti. Prvá časť (začiatok prefixu) je zarovnaná na najnižšiu dĺžku prefixov v triede a uložená v hašovacej tabuľke. Teda napríklad pre triedu s dĺžkami 8 až 17 je ako prvá časť označených prvých 8 bitov prefixu. Na uloženie druhej časti (dokončenie prefixu) je použitý TBM strom. Pre každý začiatok z hašovacej tabuľky je vytvorený vlastný strom obsahujúci všetky platné dokončenia tohto začiatku.

Popísané delenie prefixov na začiatky a dokončenia je naznačené na obrázku 4.1. Červenou farbou je označený prefix rozdelený na začiatok a koniec. Vodorovné čiarkované čiary predstavujú zarovnanie dĺžok začiatkov prefixov v daných triedach. Najdlhšia trieda je zarovnaná rovno na dĺžku IP adresy a neexistujú pre ňu dokončenia. Na obrázku je vidno aj realizáciu dohľadania dokončenia prefixu algoritmom TBM na strome nízkej výšky. Ako už bolo spomínané, priepustnosť algoritmu TBM je na nízkom strome relatívne vysoká. Vyhľadanie načrtnutého prefixu bude trvať len 3 kroky – výber z hašovacích tabuliek a 2 kroky TBM. Z pohľadu TBM sa dá navrhnutý koncept chápať ako zníženie prechádzanej výšky stromu pomocou preskakovania vrchných poschodí. Preskakovanie je realizované pomocou dopredu vypočítaných priamych odkazov na uzly nižších poschodí.



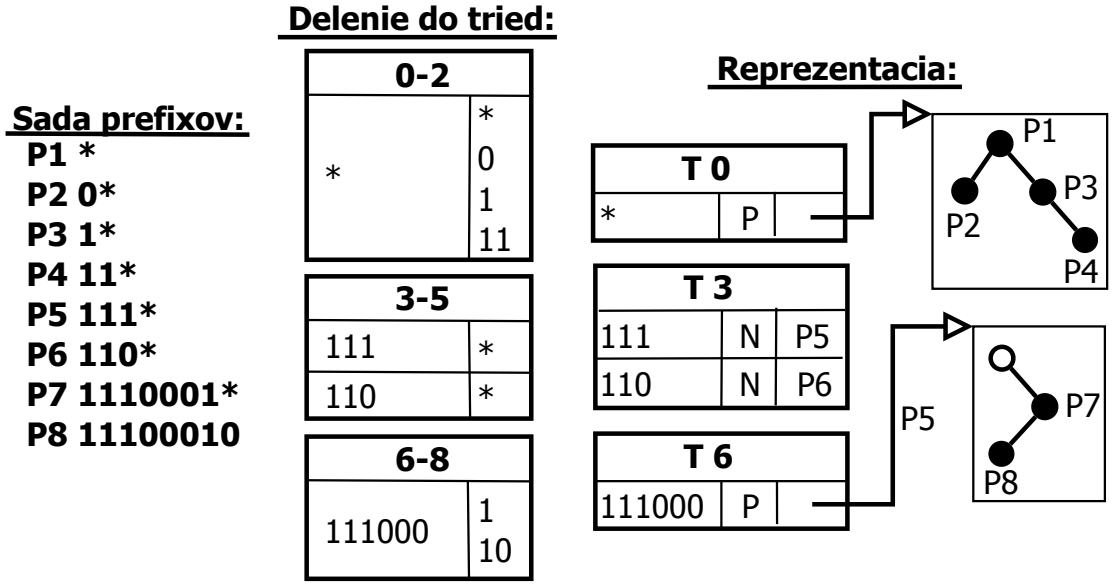
Obr. 4.1: Delenie prefixov na hašované začiatky a dokončenia v TBM

Pri delení prefixov na začiatok a dokončenie nie je nutné každému začiatku tvoriť TBM strom. Pre najkratšie prefixy v triede je dokončenie prázdne. Ak teda neexistuje iný prefix v triede s rovnakým začiatkom, obsahoval by TBM strom priradený tomuto začiatku len koreňový uzol. Efektívnejšie je preto priamo v hašovacej tabuľke označiť či je daný začiatok sám o sebe prefixom, ktorý nemá v rámci triedy žiadne dokončenie (predĺženie). Preto je potrebné definovať značky podobne ako v algoritme BSP. Značka v tomto prípade hovorí, či daný začiatok z hašovacej tabuľky pokračuje dokončením v TBM.

Pri delení prefixov na začiatok a dokončenie vzniká problém so začiatkami, ktoré sami nie sú platnými prefixmi. Pri nich môže dôjsť k situácii, že pre vstupnú adresu a ku nej vybratý zhodný začiatok prefixu, neexistuje zhodné dokončenie prefixu. To by si vynútilo opätovné hľadanie existencie kratšieho zhodného začiatku, ktorý by mohol zodpovedať síce kratšiemu, ale zhodnému prefixu vstupu. Je však možné použiť podobné riešenie ako v BSP algoritme. Pre každý začiatok, ktorý nie je sám o sebe platným prefixom, bude dopredu nájdený a uložený jeho najdlhší zhodný prefix medzi kratšími prefixmi v sade, ten bude slúžiť ako výsledok v prípade nenájdenia zhody s dokončeniami daného začiatku.

Ďalšou možnosťou vylepšenia je nepotrebnosť identifikátora prefixu a identifikátoru dokončenia TBM naraz v jednom zázname tabuľky. V každom je potrebný len jeden z nich a zároveň to, ktorý je rozlíšené značkou v danom zázname. Pri záznamoch so značkou sa jedná o identifikátor dokončenia, pri záznamoch bez značky o identifikátor prefixu. Použitím spoločného prvku v zázname pre oba identifikátory je veľkosť každého záznamu hašovacej tabuľky zmenšená o veľkosť jedného identifikátora.

Na obrázku 4.2 je možné vidieť reprezentáciu jednoduchej sady prefixov. V zobrazenom príklade bolo zvolené delenie do tried prefixov s rozsahom dĺžky 3 a na dokončenia bol využitý TBM so stride 3. Prefixy sú v prvom kroku rozdelené do tried a zároveň sú aj rozdelené na začiatky a dokončenia. V príklade je vidno, že prefixy P1 až P4 (alebo P7 a P8) patria do rovnakej triedy a zdieľajú v nej rovnaký začiatok. Preto je tento začiatok uvedený len raz a dokončenia sú pri ňom spojené. V druhom kroku je vytvorená reprezentácia dokončení Tree Bitmap uzlov (pre prehľadnosť boli ponechané v tvare Trie). Čísla v názve tabuľky reprezentujú dĺžku v nej ukladaných začiatkov. V tabuľke T3 vidno dva prefixy dĺžky 3, ku ktorým neexistuje pokračovanie (sú označené N) a teda pre ne nie je vytvorený TBM. V tabuľke T0 je začiatok dĺžky 0 a má pokračovania, z pokračovaní je teda vytvorený TBM. Tento začiatok je zároveň platným prefixom (v koreni vytvoreného TBM je platný prefix). Na rozdiel od toho začiatok v T6 je pokračujúcim, ale sám o sebe nie je prefixom. Pri šípke smerujúcej ku TBM je teda poznačený dopredu vypočítaný najdlhší zhodný prefix tohto začiatku (P5), ktorý sa využije v prípade zhody vstupu so začiatkom, ale nenájdenia zhody v pokračovaniach. Pri začiatku s tabuľky T0 prípad nenájdenia zhody nastať nemôže, preto jeho najdlhší zhodný prefix nie je nutné dopredu vypočítať.



Obr. 4.2: Vytvorenie reprezentácie sady prefixov HASH-TBM algoritmom

Hľadanie najdlhšieho zhodného prefixu pre vstupnú adresu v popísanej reprezentácii pozostáva z dvoch krokov. V prvom je pristúpené do všetkých atomických hašovacích tabuliek naraz. Z výsledkov je vybraný najdlhší zhodný začiatok, pri správnom naplnení existuje maximálne jeden. Ak nemá vybratý začiatok značku pokračovania alebo ak nebol nájdený žiaden zhodný začiatok, vyhľadávanie končí. V druhom kroku sa pokračuje pre začiatky so značkou. Druhý krok pozostáva s dohľadania zhodného pokračovania prefixu v TBM strome. Využíva sa štandardný postup TBM popísaný v 2.4. Pri nenájdení zhody v pokračovaniach je ako výstup použitý uložený najdlhší zhodný prefix začiatku.

4.3 Optimalizácie

Algoritmus uvedený v predošlej časti je možné v niektorých prípadoch ďalej optimalizovať. V tejto časti sú predstavené 3 možné optimalizácie algoritmu Hash Tree Bitmap vedúce k zníženiu spotreby zdrojov jednotky, zmenšeniu pamäťovej náročnosti a tiež k zrýchleniu výpočtu. Všetky z uvedených optimalizácií boli využité aj pri implementácii výslednej architektúry.

Prvá z možných optimalizácií sa týka tried s krátkymi prefixmi. Počet všetkých rôznych prefixov dĺžky n sa vypočíta ako 2^n. Pre krátke prefixy je toto číslo relatívne malé. Ak je menšie ako veľkosť atomickej hašovacej tabuľky použitej v architektúre, je zbytočné prideľovať takejto skupine celú tabuľku. Napríklad v architektúre s veľkosťou atomickej tabuľky 1024 položiek sa pri uložení prefixov dĺžky 4, ktorých existuje $2^4 = 16$, využije maximálne $\frac{1}{64}$ položiek v tabuľke. Pre krátke skupiny prefixov je preto výhodnejšie použiť iný spôsob ukladania.

Pri veľkosti atomickej tabuľky K je pre všetky skupiny prefixov s dĺžkou n takou, že $2^n < K$, výhodnejšie použiť tabuľku s priamym adresovaním. Ide o tabuľku, kde samotný prefix je indexom (adresou) poľa a nie je potrebné žiadne hašovanie. Popísaný prístup sa podobá princípu využívanému v algoritme rozšíreného Trie. Dopredu je vybratá vhodná hodnota N, ako maximálna dĺžky prefixov, pre ktoré sa použije tabuľka s priamym prístupom. Veľkosť tejto tabuľky bude 2^N prvkov. Všetky prefixy s dĺžkou $n < N$ budú následne rozšírené na dĺžku N a spoločne uložené v tejto tabuľke.

Rozšírením prefixu dĺžky $n < N$ na dĺžku N sa rozumie jeho nahradenie všetkými takými prefixmi dĺžky N, že nahradzovaný prefix je ich začiatkom. Inak povedané, ku nahradzovanému prefixu sú pridané všetky možné kombinácie núl a jednotiek tak, aby vznikli prefixy požadovanej dĺžky. Napríklad pri rozšírení prefixu 01* na dĺžku 4 dostaneme štyri nové prefixy 0100*, 0101*, 0110* a 0111*.

Popísaným spôsobom je možné reprezentovať všetky prefixy kratšie a rovnako dlhé ako zvolené N pomocou jedinej tabuľky s priamym prístupom. Do nej sú tiež pridané začiatky dlhších prefixov vytvorené ich delením popísaným v algoritme. Navyše vďaka priamemu adresovaniu prefixmi, má každé políčko tejto tabuľky jednoznačne priradený svoj kľúč (prefix je adresou aj kľúčom). V tabuľke s priamym adresovaním teda nie je potrebné ukladať kľúče dát v záznamoch ako pri hašovacej tabuľke. Vo výsledku je ušetrená pamäť a zároveň nepotrebnosť hašovania šetrí aj zdroje na čipe.

Ďalšou možnosťou optimalizácie je zníženie počtu hašovacích jednotiek. Hašovacia funkcia je všeobecne zložitejší výpočtový proces vyžadujúci značnú plochu na čipe. Každá atomická hašovacia tabuľka má vlastnú hašovaciu jednotku. Pretože pri hašovaní sa používa tvar vstupnej hodnoty vymaskovaný na potrebnú dĺžku prefixov v tabuľke. Pre každú tabuľku môže byť nastavená iná dĺžka a preto aj výsledkom pre daný vstup bude iná hodnota hašu. Zároveň ale pre tabuľky s rovnakou dĺžkou je hodnota hašu, pre rovnaký vstup, rovnaká.

V prípade, že je v architektúre viac atomických tabuliek ako používaných tried prefixov je vždy väčší počet tabuliek nastavených na rovnakú dĺžku. Preto je zbytočné pre každú z nich osobitne rátať hodnotu hašovacej funkcie. Jednoduchšie je vypočítať hodnotu hašu vstupnej hodnoty pre každú z tried prefixov. Následne v každej tabuľke už len vybrať z vypočítaných hodnôt tú, ktorá zodpovedá dĺžke konkrétnej tabuľky. Proces výberu je výpočtovo podstatne jednoduchšia úloha ako hašovanie, preto je jej náročnosť na zdroje na čipe nižšia.

Napríklad v architektúre, v ktorej sú IPv4 adresy delené do tried s rozsahom dĺžky 8 vznikne 5 tried prefixov (0-7, 8-15, 16-23, 24-31 a 32). Pri použití tabuľky s priamym prístupom pre dĺžku prefixov 8 je hašovanie využité len v posledných troch z nich (prefixy skupiny 0-7 sú rozšírené a 8-15 sú rozdelené na začiatky a dokončenia). Teda pre každý vstup existujú len tri rôzne použiteľné hodnoty hašu. Ak takáto architektúra potom obsahuje 16 atomických hašovacích tabuliek, je popísanou optimalizáciou znížený potrebný počet hašovacích jednotiek zo 16 na 3. Naviac je ale potrebných 16 jednotiek realizujúcich výber jednej hodnoty z troch, tie sú ale podstatne jednoduchšie a menej náročné na zdroje ako hašovacie jednotky. Vo výsledku je teda náročnosť architektúry na plochu čipu nižšia.

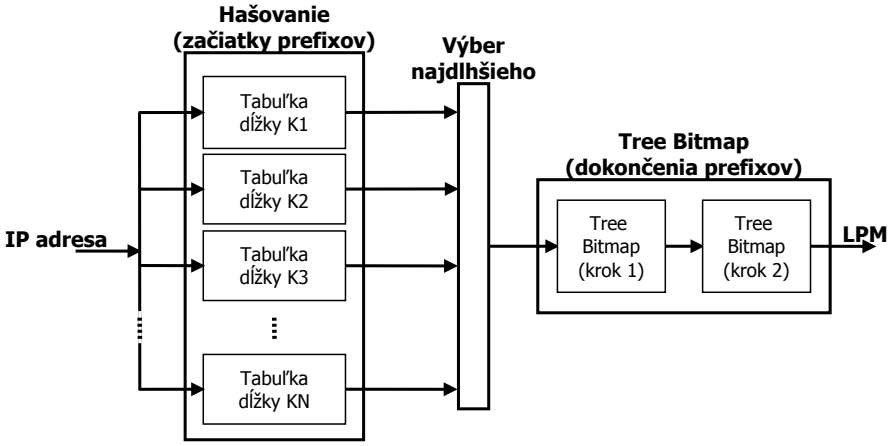
Posledná optimalizácia sa týka dohľadania dokončení prefixov pomocou Tree Bitmap algoritmu. V prípade, že nastavené parametre architektúry vyžadujú viacposchodový TBM strom je potrebných viac krokov algoritmu na prechod stromom. Pri použití jednej výpočtovej TBM jednotky by bol potrebný sekvenčný prechod stromom, čo by viedlo k spomaleniu spracovania vstupných adries.

Výhodnejším riešením je použitie viacerých TBM jednotiek zapojených za sebou. Konkrétne pre každé poschodie TBM stromu jednu. Potom každá z nich spracuje vždy len jedno z poschodí stromu a výsledky odovzdá nasledujúcej, ktorá spracuje ďalšie poschodie. Takto zostavená výpočtová línia je potom schopná prechodu TBM stromom v konštantnom čase bez nutnosti sekvenčného prechodu. Výsledkom je teda zvýšenie priepustnosti architektúry za cenu vyšších nárokov na plochu čipu.

Zvýšenie nárokov na plochu čipu však môže byť čiastočne kompenzované úpravou poslednej TBM jednotky, najmä ak sú v architektúre použité len dve TBM jednotky. Pre všetky TBM uzly spracovávané poslednou jednotkou platí, že sú vždy listovými uzlami, teda určite nemajú následníkov. Nie je preto potrebné uchovávať externú bitmapu týchto uzlov. Zároveň nie je potrebné v poslednej TBM jednotke implementovať logiku súvisiacu s výpočtom následníka. Popísaným zjednodušením poslednej TBM jednotky je znížená jej náročnosť na plochu čipu a zároveň sú znížené pamäťové nároky na uchovanie TBM uzlov posledného poschodia.

4.4 Implementácia

Logická štruktúra implementovanej jednotky zodpovedá logickej štruktúre algoritmu naznačenej na obrázku 4.3. Vyhľadanie začína výberom zhodných začiatkov prefixov z paralelných hašovacích tabuliek. Následne je vybratý najdlhší spomedzi zhodných začiatkov, pre ktorý je dohľadané dokončenie pomocou Tree Bitmap algoritmu. Zobrazená logická schéma je realizovaná architektúrou a jej časti je možné rozpoznať aj v nasledujúcich schémach štruktúry implementácie architektúry.

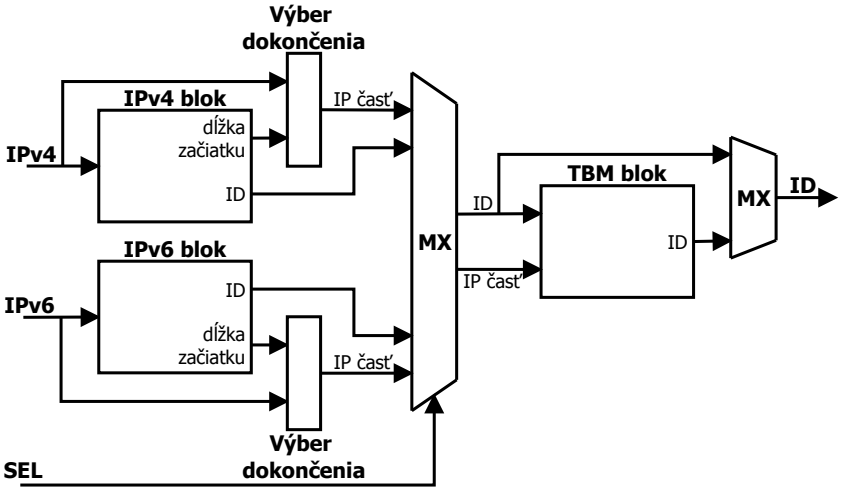


Obr. 4.3: Logická schéma HW algoritmu LPM

V tabuľke 4.1 sú popísané signály tvoriace rozhranie implementovanej architektúry. Implementovaná bola architektúra celkovo v 2 verziách, ktoré sa navzájom líšia počtom podporovaných IPv4 prefixov a teda veľkosťou využitej pamäte čipu. Verzie sú označené ako FULL (plná verzia) a LITE (odľahčená verzia). Pre obe implementované verzie ostáva rozhranie rovnaké.

Signál	Šírka	Popis
Vstup		
CLK	1	globálny hodinový signál FPGA
RESET	1	globálny signál reštartovania
CONTEXT	1	kontext vstupnej adresy
SEL	1	výber medzi IPv4 a IPv6 adresou (0 pre IPv4 a 1 pre IPv6)
IPv4	32	vstupná IPv4 adresa
IPv6	128	vstupná IPv6 adresa
VAL_IN	1	platnosť vstupných dát
Výstup		
VAL_OUT	1	platnosť výstupných dát (oneskorený signál VAL_IN)
FIND	1	informácia o úspešnosti nájdenia prefixu
ID	16	identifikátor nájdeného najdlhšieho zhodného prefixu
Plnenie		
MI32	mi32	štandardné pamäťové rozhranie pre zápis

Tabuľka 4.1: Signály rozhrania HW architektúry

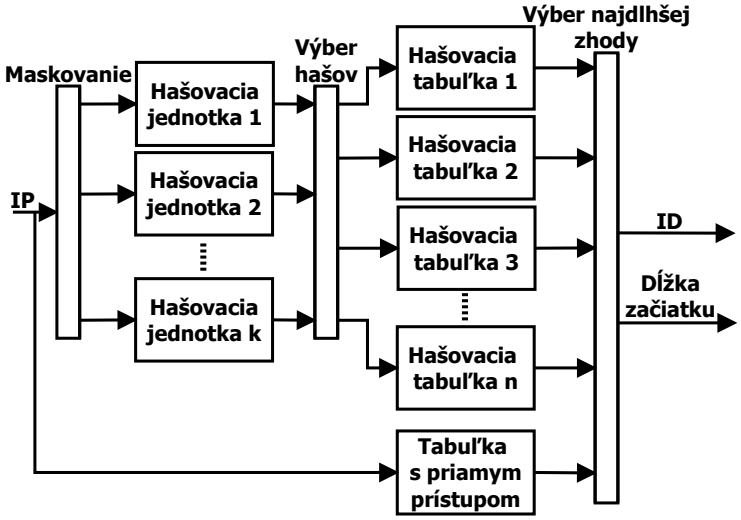


Obr. 4.4: Logická schéma implementácie HW architektúry

Na obrázku 4.4 je možné vidieť základnú logickú štruktúru implementovanej jednotky hľadania najdlhšieho zhodného prefixu. Vyhľadávanie začína nájdením najdlhšieho zhodného začiatku prefixu adries v IPv4 a IPv6 bloku. Výstupom sú informácie o identifikátore nájdeného záznamu alebo adrese pokračovania (ID) a dĺžka zhodného začiatku. Z hodnoty ID je možné určiť jeho typ, teda ID v sebe obsahuje aj značku pokračovania prefixu. Na základe dĺžky zhodného začiatku je z IP adresy vybratých 7 bitov (IP časť) na hľadanie pokračovania prefixu. Z výsledkov blokov IPv4 a IPv6 je vybratý jeden podľa signálu SEL, určujúceho aký typ IP adresy sa vyhľadáva.

Výsledky získané v prvej časti vyhľadávania slúžia ako vstupy bloku TBM realizujúceho dohľadanie zhodného dokončenia prefixu. Dohľadanie pozostáva z dvoch krokov TBM algoritmu so $stride = 4$. Výstupom je identifikátor nájdeného dokončenia (ID). Z výstupu TBM bloku a výstupu IP bloku je nakoniec určený celkový výstupu.

Logickej schéme na obrázku 4.4 zodpovedá podrobnejšia implementačná schéma B.1.



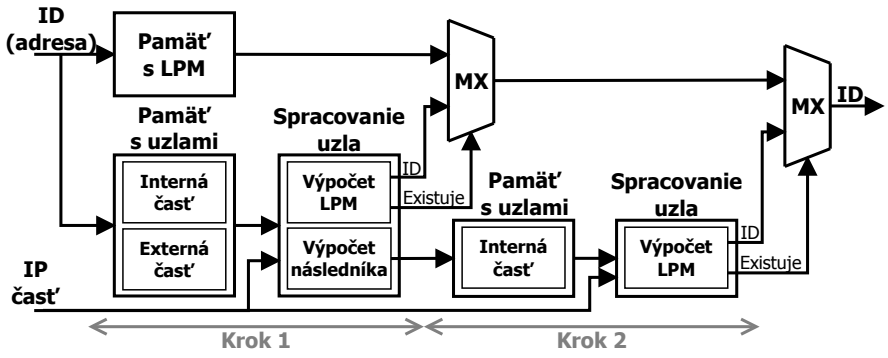
Obr. 4.5: Logická schéma implementácie IP bloku

Na obrázku 4.5 je zobrazená logická štruktúra implementácie IP blokov. IP blok pracuje s delením prefixov do tried podľa dĺžky s rozsahom 8. V schéme je možné vidieť optimalizáciu pridaním tabuľky s priamym prístupom pre triedy prefixov s dĺžkou začiatku do 8 (triedy 0-7 a 8-15). Vidno tiež osamostatnené hašovacie jednotky mimo atomických hašovacích tabuliek a s tým súvisiaci výber správneho hašu pre tabuľky. Oddelenie hašovacích funkcií od tabuliek má význam len pre $k < n$, ako bolo popísané v predošlej sekcii. V IPv4 bloku je k maximálne 3 a použité n je 8 alebo 18, preto sa využíva zobrazené delenie. V IPv6 bloku je k maximálne 31 a použité n je len 4, preto sú hašovacie jednotky priamo súčasťou tabuliek a nie je potrebný ani výber hašov.

Hašovacie tabuľky majú veľkosť 1024 položiek pre IPv4 blok a 512 položiek pre IPv6 blok. Každá tabuľka realizuje vyhľadanie vstupnej IP adresy na pozícii danej hašom a výsledkom je informácia o nájdení zhody. Ku každému uloženému začiatku prefixu v hašovacej tabuľke sú uložené aj ďalšie metadáta. Metadáta sú tvorené hlavne identifikátorom patriacim k prefixu, ktorý v sebe obsahuje aj informáciu o pokračovaní prefixu TBM dokončením. Pre tabuľku s priamym prístupom postačuje ukladať len metadáta, lebo horný bajt IP adresy tvorí jednoznačnú adresu do tabuľky.

Nakoniec je z výsledkov všetkých tabuliek vybratý indentifikátor patriaci najdlhšiemu z nájdených zhodných začiatkov. Implementovaný je prioritným výberom, kedy je vždy vybratý prvý zhodný začiatok. Tento výber vedie na pevné pridelenie priority tabuľkám podľa ich čísla. Čím nižšie číslo tabuľky, tým vyššia priorita výsledku. Aby bol vždy vybraný najdlhší zhodný začiatok je nutné správne prideľovať dĺžky atomickým hašovacím tabuľkám. Teda ako najdlhšej je nutné nastaviť tabuľky s najnižším číslom (najvyššou prioritou) a opačne. Tabuľka s priamym prístupom reprezentujúca najkratšie prefixy má implicitne pridelené najvyššie číslo (najnižšiu prioritu).

Podrobnejšie schémy implementácie oboch IP blokov je možné nájsť v prílohe na obrázkoch B.2 (IPv4) a B.3 (IPv6).



Obr. 4.6: Logická schéma implementácie TBM bloku

Logická schéma TBM bloku je na obrázku 4.6. Blok slúži na dohľadanie dokončení prefixov pomocou algoritmu Tree Bitmap. Maximálna dĺžka dokončení je 7 bitov a dohľadanie je implementované ako dva kroky algoritmu TBM s parametrom $stride = 4$. Na obrázku je možné vidieť, že jednotka realizujúca druhý krok výpočtu (vpravo dole) neobsahuje výpočtovú logiku ani pamäť potrebnú pri výpočte následníka.

Hľadanie dokončenia prefixu začína výberom dát TBM uzla prvého kroku z pamätí podľa vstupného identifikátora patriaceho začiatku prefixu, identifikátor tvorí adresu do pamäte. Interná časť obsahuje informácie potrebné na výpočet najdlhšieho zhodného dokončenia prefixu v rámci aktuálneho TBM uzla, konkrétne internú bitmapu a identifikátor prvého

prefixu. Externá časť obsahuje informácie potrebné na výpočet následníka, konkrétne externú bitmapu a identifikátor (adresu) prvého následníka. Dáta o uzle, získané z pamäte, sú privedené na vstup výpočtovej jednotky implementujúcej nájdenie najdlhšieho zhodného prefixu v TBM uzle a výpočet následníka podľa privedenej časti IP adresy. K prvému kroku TBM algoritmu je pridaná pamäť obsahujúca dopredu vypočítaný výsledok najdlhšieho zhodného prefixu začiatku, ktorému patrí dané pokračovanie.

V druhom kroku je vždy spracovávaný uzol druhého poschodia TBM stromu, vybraný podľa výsledku jednotky na výpočet následníka v prvom kroku. Ako už bolo spomínané, uzol druhého kroku obsahuje len pamäť a výpočtovú jednotku potrebnú pre nájdenie najdlhšieho zhodného prefixu v TBM uzle.

Z výsledkov vyhľadania najdlhšieho zhodného prefixu v jednotlivých krokoch a z pamäte s vopred vypočítaným implicitným výsledkom je určený celkový výsledok. V každom kroku TBM algoritmu je ako medzivýsledok ukladaný najdlhší doteraz nájdený zhodný prefix. Pamäť s vopred vypočítanými výsledkami teda nahradzuje medzivýsledok, ktorý by bol získaný predošlými krokmi TBM algoritmu, preskočenými hašovacími tabuľkami. Po každom kroku je teda, podľa úspešnosti nájdenia pokračovania, označený za medzivýsledok buď novozískaný identifikátor alebo je ponechaný pôvodný z minulého kroku.

Podrobnejšiu štruktúru implementovaného TBM bloku je možné nájsť v prílohe na obrázku B.4.

Vo všetkých zobrazených schémach sú zakreslené len dátové cesty a bloky súvisiace priamo s vyhľadávaním najdlhšieho zhodného prefixu. Cesty a bloky súvisiace s plnením neboli kvôli prehľadnosti schém zakreslené. Podrobnejší popis plnenia jednotky konfiguračnými dátami spolu s popisom adresného priestoru je možné nájsť v prílohe A.

Vyhľadávať najdlhší zhodný prefix adries je v implementovanej architektúre možné v jednom z dvoch rôznych kontextoch (signál CONTEXT). Kontextovo závislé vyhľadávanie bolo implementované z dôvodu možnosti bezstratového prepnutia konfiguračných dát za behu. Teda pokým vyhľadávanie pracuje v jednom kontexte, môžu byť konfiguračné dáta druhého zatiaľ zmenené bez vplyvu na výsledky vyhľadávania. Po skončení zmien je v jednom takte prepnutý kontext a vyhľadávanie hneď pokračuje v novom kontexte.

Kapitola 5

Výsledky

5.1 Priepustnosť na viacjadrovom stroji

Všetky algoritmy boli testované na viacerých rôzne veľkých prefixových sadách. Prefixové sady boli vytvorené z reálneho súboru prefixov získaného zo smerovača. Sú to prefixy zo smerovacej tabuľky bgptable, ktorá pozostávala z 322 684 prefixov. Jej vhodným delením boli vytvorené menšie prefixové sady rôznej veľkosti. Okrem reálnych prefixových sád boli použité aj umelo vytvorené sady označené synth.

Syntetické sady boli vytvorené použitím programu makeset (makeset.c). Program makeset generuje sadu obsahujúcu daný počet prefixov dĺžky 32 rovnomerne rozložených na celý adresný priestor IPv4 adries. Vďaka maximálnej dĺžke všetkých prefixov predstavujú umelo vytvorené sady najhorší prípad pre algoritmy TBM a SST z hľadiska výkonnosti pri vyhľadávaní najdlhšieho zhodného prefixu. Na dosiahnutie najhoršieho prípadu pre algoritmus BSP bolo do každej z umelo vytvorených sád pridaných 31 prefixov, po jednom pre každú z dĺžok 1 až 31. To vedie k vzniku maximálneho počtu tried prefixov v BSP.

Prefixová sada	**Počet prefixov**	**Rôznych dĺžok prefixov**	**Priemerná dĺžka prefixu**	**Potrebných uzlov Trie**
bgptable	1 009	14	22,45	12 832
	10 084	19	22,48	86 281
	53 781	23	22,46	296 929
	161 342	23	22,47	596 223
	322 684	24	22,47	842 262
synth	1 031	32	31,52	24 023
	10 031	32	31,95	206 383
	50 031	32	31,99	915 535
	150 031	32	32,00	2 512 143
	300 031	32	32,00	4 724 287
	500 031	32	32,00	7 524 287

Tabuľka 5.1: Vlastnosti použitých prefixových sád

Pre rozloženie prefixov generované programom makeset predstavuje hašovanie využívajúce horné bity IP adresy dokonalú hašovaciu funkciu. Pri použitej veľkosti hašovacích tabuliek, najbližsia mocnina 2 vyššia ako počet záznamov v tabuľke, tvorí bezkolízne mapo-

vanie prefixov syntetických sád do hašovacích tabuliek. Zároveň je hašovacia funkcia výberu horných bitov najjednoduchšou z hľadiska zložitosti výpočtu. Dosiahnuté priepustnosti algoritmu BSP na syntetických sadách s použitím horných bitov ako hašu, budú teda určovať teoretické maximum dosiahnuteľné algoritmom BSP pri maximálnom počte tried. V reálnej konfigurácii je dosiahnutie takejto ideálnej situácie nemožné, namerané výsledky popísaných konfigurácii preto nebudú použité pri celkovom vyhodnotení výkonností algoritmov.

Základné vlastnosti použitých prefixových sád je možné vidieť v tabuľke 5.1. Testovaných bolo 5 reálnych prefixových sád a 6 umelo vytvorených sád. Stĺpec tabuľky s počtom rôznych dĺžok prefixov v sade je významný pre algoritmus BSP. Pre algoritmy TBM a SST sú dôležité stĺpce s priemernou dĺžkou prefixov v sade a s veľkosťou Trie. Z tabuľky je možné vidieť, že na reprezentáciu umelo vytvorených prefixových sád je potrebný niekoľkonásobne väčší Trie ako na reprezentáciu podobne veľkých reálnych sád.

Testovanie prebiehalo na počítači s operačným systémom Linux, konkrétne verzia Red Hat Enterprise Linux Server release 5.1. Počítač bol vybavený dvoma procesormi Intel Xeon Processor X5450 s výkonnostnými parametrami [17]:

- **Počet jadier:** 4
- **Počet vlákien:** 4
- **Frekvencia hodín:** 3 GHz
- **Frekvencia zbernice:** 1333 MHz
- **L2 cache:** 12 MB

Celkovo preto testovací počítač disponoval 8 jadrami s frekvenciou 3 GHz a podporoval súbežný beh 8 vlákien.

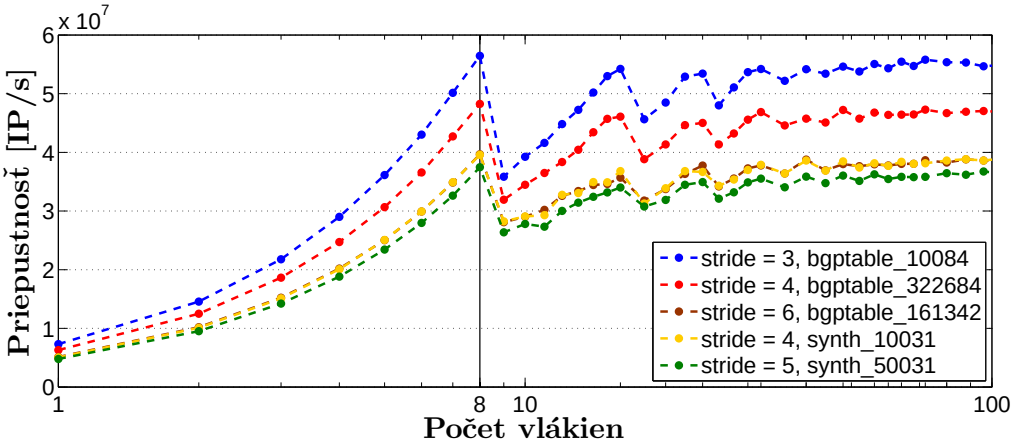
Konfiguračné súbory obsahujúce reprezentácie prefixových sád potrebné pre algoritmy boli vytvorené dopredu s použitím Netbench frameworku. Podrobnejší popis je možné nájsť v predošlej kapitole. Vstupné dáta (IP adresy) pre hľadanie najdlhšieho zhodného prefixu boli vytvorené umelo, s využitím programu makebin (makebin.c). Program makebin z textového súboru prefixov generuje binárny súbor obsahujúci IPv4 adresy. Vytvorený binárny súbor obsahuje daný počet náhodne zoradených adries, rovnomerne rozložených vzhľadom na prefixy zo vstupného súboru. Inak povedané, každému prefixu je vytvorený rovnaký počet zodpovedajúcich IP adries, ktoré sú potom náhodne usporiadané do výsledného súboru. Pre testovanie boli vytvorené súbory obsahujúce po 10 000 000 IP adries.

Algoritmus Tree Bitmap bol testovaný pre štyri rôzne konfigurácie líšiace sa použitou hodnotou stride. Testované boli hodnoty 3, 4, 5 a 6. Stride ovplyvňuje výšku TBM stromu a čas potrebný na spracovanie jedného TBM uzla. Výšku stromu ovplyvňuje nepriamoúmerne, čím urýchľuje vyhľadávanie. Na druhej strane čas potrebný na spracovanie jedného TBM uzla ovplyvňuje nepriaznivo. Závislosť dátovej šírky bitmáp v TBM uzle na stride je exponenciálna a bitmapy sú spracovávané po bitoch, preto exponenciálne rastie čas spracovania jedného TBM uzla v závislosti na stride.

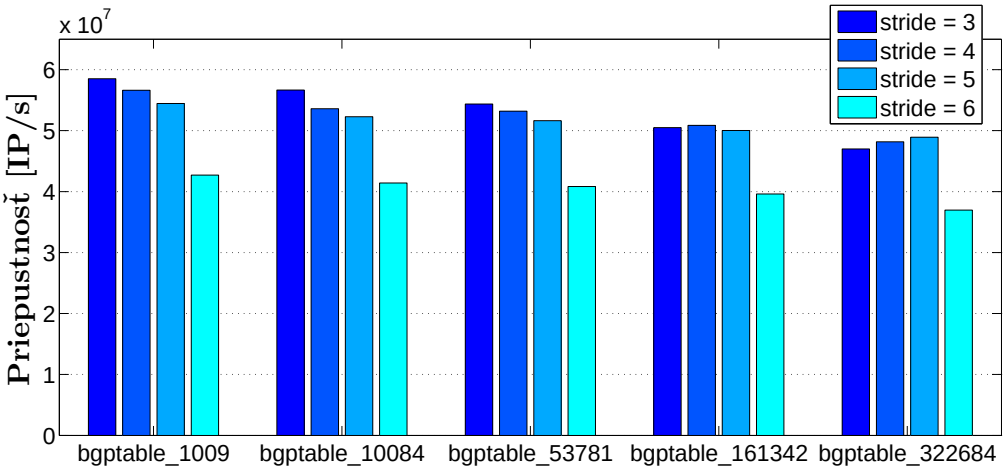
Závislosť priepustnosti algoritmu Tree Bitmap na počte vlákien je možné vidieť v grafe na obrázku 5.1. Zobrazený graf ukazuje závislosť priepustnosti v počte IP adries za sekundu vzhľadom na počet vlákien použitých pri výpočte. Každá čiara reprezentuje jednu konfiguráciu. Z grafu vidno, že priepustnosť algoritmu Tree Bitmap je najlepšia pre počet vlákien 8, čo je počet súbežných vlákien podporovaných na testovacom počítači.

Závislosť priepustnosti algoritmu Tree Bitmap na použitej hodnote stride a na veľkosti prefixovej sady je zachytená v grafoch na obrázkoch 5.2 (reálne sady) a 5.3 (syntetické sady).

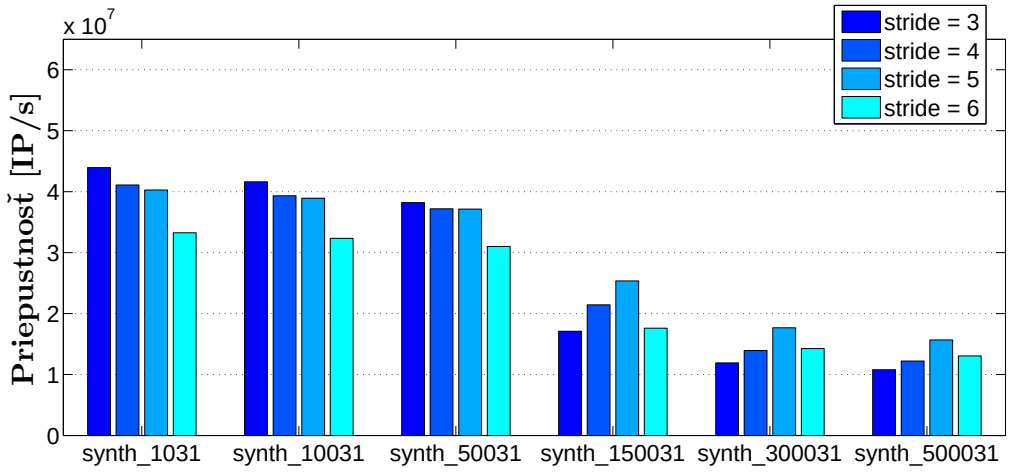
Hodnoty boli namerané pri použití 8 vlákien, ako priemer z 10 meraní. Z grafov je možné vidieť, že hodnota stride aj veľkosť prefixovej sady ovplyvňujú priepustnosť algoritmu. S rastúcou prefixovou sadou klesá priepustnosť. Nižšiu priepustnosť dosahuje algoritmus aj na syntetických sadách predstavujúcich najhorší prípad. Zaujímavá je aj postupná zmena najvýkonnejšej konfigurácie TBM. S rastúcou prefixovou sadou sa stávajú najvýkonnejšími postupne konfigurácie s vyššou hodnotou stride, resp. ich výkonnosť klesá s rastúcou prefixovou sadou pomalšie.



Obr. 5.1: Graf závislosti priepustnosti TBM na počte vlákien



Obr. 5.2: Graf priepustnosti TBM na reálnych prefixových sadách



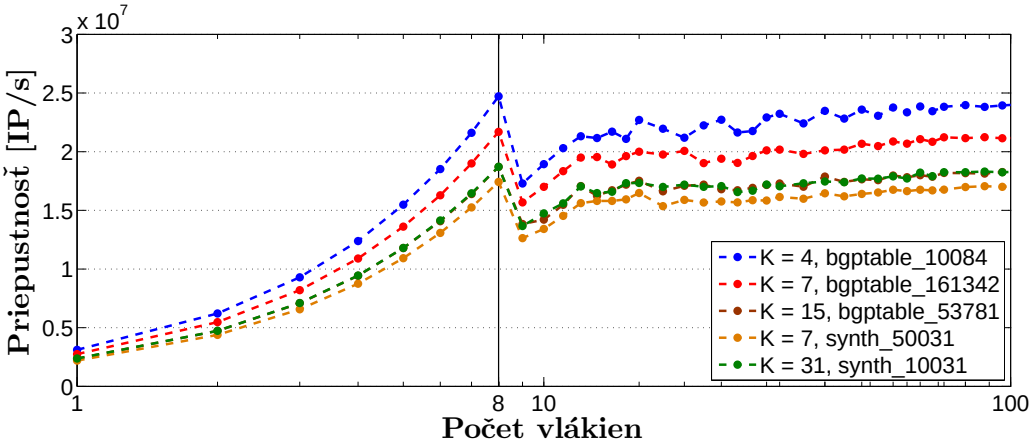
Obr. 5.3: Graf priepustnosti TBM na syntetických prefixových sadách

Algoritmus Shape-Shifting Trie bol testovaný pre štyri rôzne konfigurácie líšiace sa použitou hodnotou K. Testované boli hodnoty 4, 7, 15 a 31. Hodnota K ovplyvňuje výšku SST stromu a čas potrebný na spracovanie jedného SST uzla. Výška stromu s rastúcim K postupne klesá, čím sa urýchľuje vyhľadávanie. Na druhej strane čas potrebný na spracovanie jedného SST uzla ovplyvňuje nepriaznivo. Závislosť dátovej šírky bitmáp v SST uzle na hodnote K je lineárna a bitmapy sú spracovávané po bitoch, preto lineárne rastie čas spracovania jedného SST uzla v závislosti na stride.

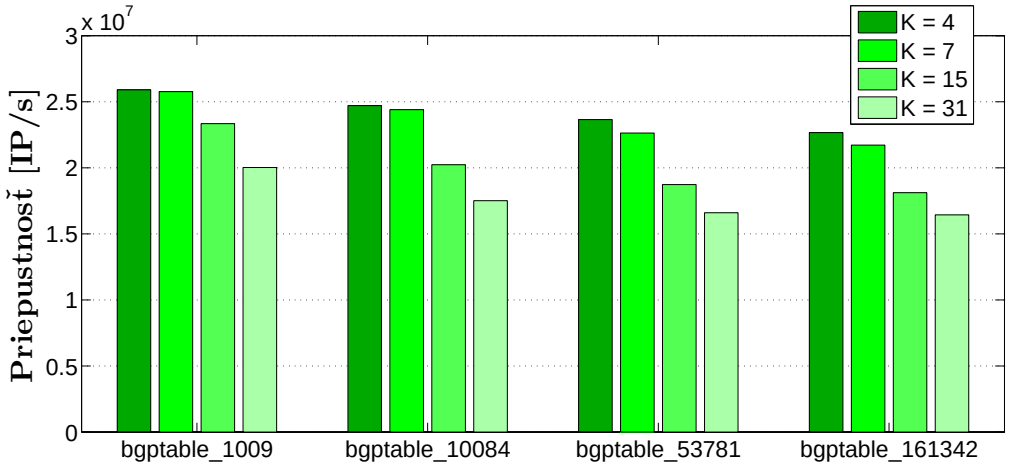
Algoritmus SST nebol testovaný pre všetky prefixové sady z dôvodu veľkej časovej náročnosti generovania konfigurácii frameworkom Netbench. Generovanie konfiguračných dát je pri algoritme SST najzložitejším spomedzi testovaných algoritmov. Zložitosť generovania teda viedla na rádovo dlhší čas vytvárania súborov s konfiguračnými dátami. Z toho dôvodu bolo možné otestovať len menšie z prefixových sád, reálne sady boli testované do veľkosti 161 342 prefixov a syntetické sady len do veľkosti 50 031 prefixov. Pre obe spomínané najväčšie sady trvalo vytvorenie jedného konfiguračného súboru rádovo 1 až 2 dni.

Závislosť priepustnosti algoritmu Shape-Shifting Trie na počte vlákien je možné vidieť v grafe na obrázku 5.4. Zobrazený graf ukazuje závislosť priepustnosti v počte IP adries za sekundu vzhľadom na počet vlákien použitých pri výpočte. Každá čiara reprezentuje jednu konfiguráciu. Z grafu vidno, že priepustnosť algoritmu SST je najlepšia pre počet vlákien 8, čo je počet súbežných vlákien podporovaných na testovacom počítači.

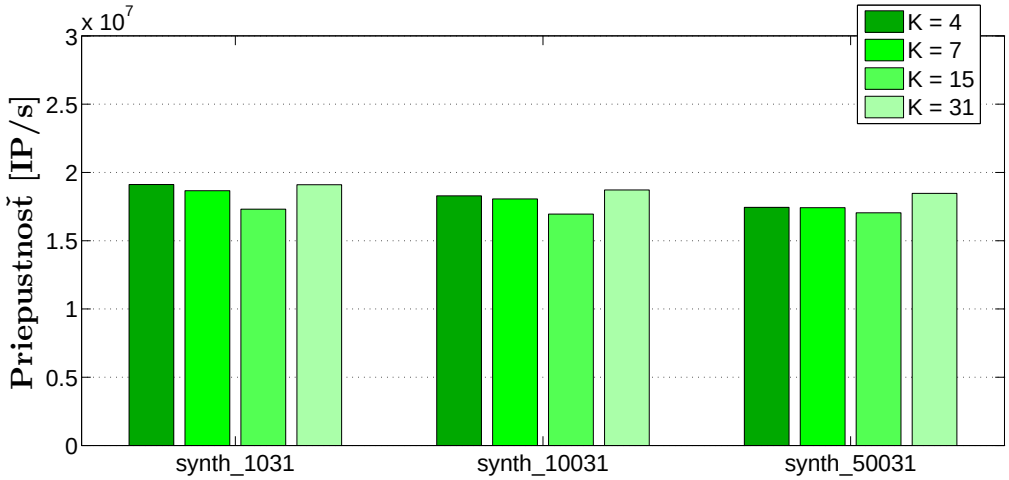
Závislosť priepustnosti algoritmu Shape-Shifting Trie na použitej hodnote K a na veľkosti prefixovej sady je zachytená v grafoch na obrázkoch 5.5 (reálne sady) a 5.6 (syntetické sady). Hodnoty boli namerané pri použití 8 vlákien, ako priemer z 10 meraní. Z grafov je možné vidieť, že hodnota K aj veľkosť prefixovej sady ovplyvňujú priepustnosť algoritmu. S rastúcou prefixovou sadou klesá priepustnosť. Nižšiu priepustnosť dosahuje algoritmus aj na syntetických sadách predstavujúcich najhorší prípad. Na reálnych sadách je vždy najvýkonnejšou konfigurácia s $K = 4$ a najslabšou konfigurácia s $K = 6$. Zaujímavé potom je, že pre syntetické sady je výkonnosť oboch variant veľmi podobná a okrem najmenšej sady je všade výkonnejší variant $K = 6$.



Obr. 5.4: Graf závislosti priepustnosti SST na počte vlákien



Obr. 5.5: Graf priepustnosti SST na reálnych prefixových sadách



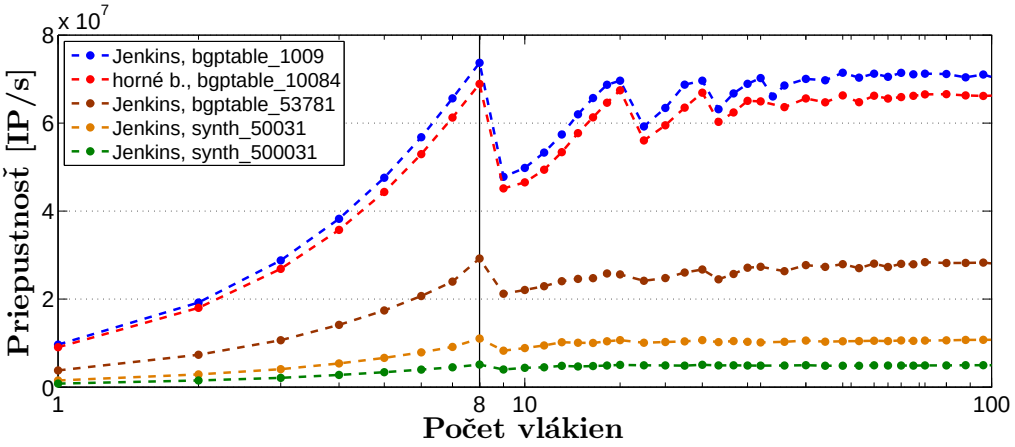
Obr. 5.6: Graf priepustnosti SST na syntetických prefixových sadách

Algoritmus Binary Search on Prefixes bol testovaný pre dve konfigurácie líšiace sa použitou hašovacou funkciou. V jednej bola použitá plnohodnotná hašovacia funkcia (Jenkins hash), v druhej priamy výber potrebného počtu horných bitov z IP adresy. Voľba hašovacej funkcie ovplyvňuje priepustnosť dvomi spôsobmi. Prvým je zložitosť výpočtu hašovanej hodnoty z IP adresy. Výpočet náročnejšej funkcie spomaľuje algoritmus viac ako jednoduchšej. Na druhej strane, kvalitná hašovacia funkcia poskytuje lepšie usporiadanie prefixov do hašovacej tabuľky. Nevhodné usporiadanie vedie k nárastu prístupového času do hašovacej tabuľky, kvôli veľkému počtu kolízií.

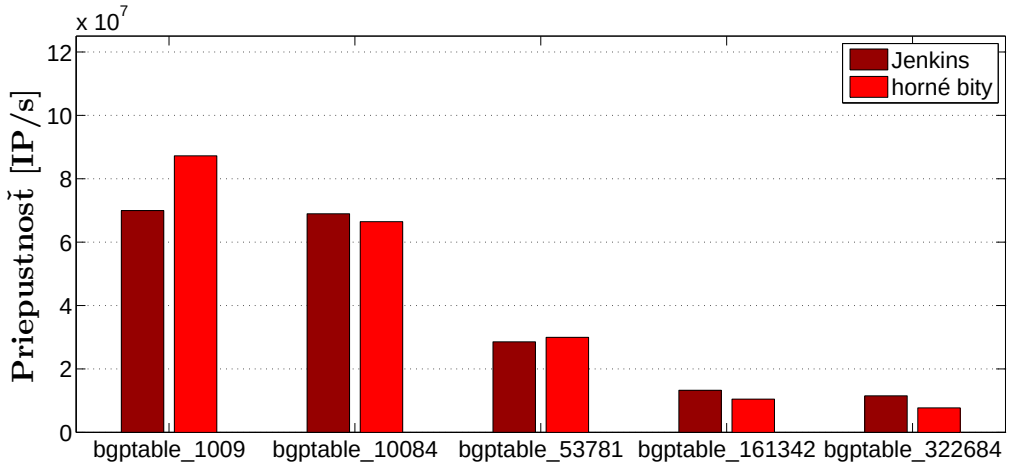
Závislosť priepustnosti algoritmu Binary Search on Prefixes na počte vlákien zobrazuje graf na obrázku 5.7. Zobrazený graf ukazuje závislosť priepustnosti v počte IP adries za sekundu vzhľadom na počet vlákien použitých pri výpočte. Každá čiara reprezentuje jednu konfiguráciu. Z grafu vidno, že priepustnosť algoritmu BSP je najlepšia pre počet vlákien 8, čo je počet súbežných vlákien podporovaných na testovacom počítači.

Závislosť priepustnosti algoritmu Binary Search on Prefixes na použitej hašovacej funkcii a na veľkosti prefixovej sady je zachytená v grafoch na obrázkoch 5.5 (reálne sady) a 5.6 (syntetické sady). Hodnoty boli namerané pri použití 8 vlákien, ako priemer z 10 meraní. Z grafov je možné vidieť, že rastúca veľkosť prefixovej sady výrazným spôsobom negatívne ovplyvňuje priepustnosť algoritmu pre reálne aj syntetické sady. Rozdiel medzi použitými spôsobmi hašovania sa v reálnych sadách veľmi neprejavuje.

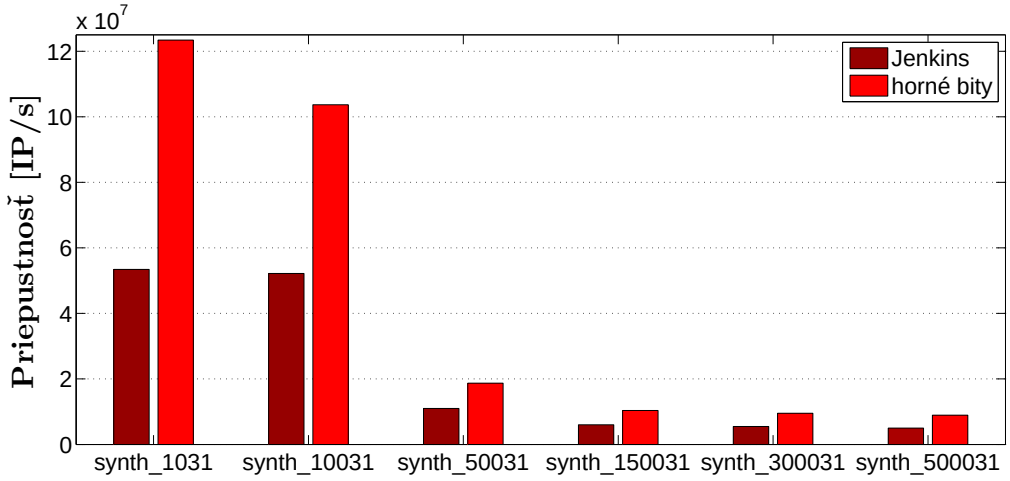
V syntetických sadách je rozdiel výkonnosti medzi použitými spôsobmi hašovania značný. Hašovanie využívajúce horné bity IP adries sa presne podľa očakávaní prejavilo na syntetických sadách ako veľmi výkonné. Nameraná hodnota priepustnosti na najmenšej syntetickej sade, 123 miliónov IP adries za sekundu, predstavuje teoretický limit dosiahnuteľnej výkonnosti algoritmu BSP na testovacom stroji. V normálnej situácii však nie je možné sa podobnému stavu priblížiť o čom svedčí aj výkonnosť dosahovaná Jenkinsnovou hašovacou funkciou a výkonnosti namerané pre reálne sady. Aj pri použití popísaného ideálneho hašovania je však viditeľný prudký pokles výkonnosti s rastom veľkosti prefixovej sady, ktorý výrazne znehodnocuje použitie algoritmu BSP.



Obr. 5.7: Graf závislosti priepustnosti BSP na počte vlákien



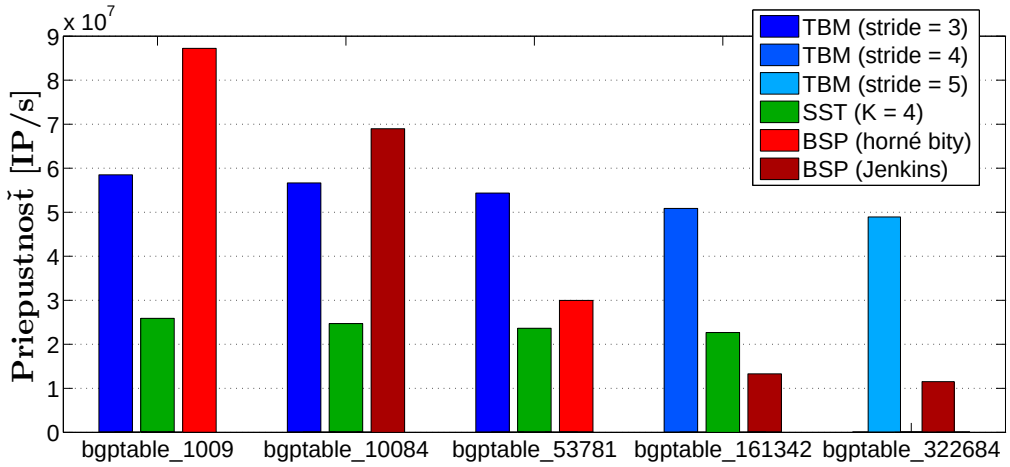
Obr. 5.8: Graf priepustnosti BSP na reálnych prefixových sadách



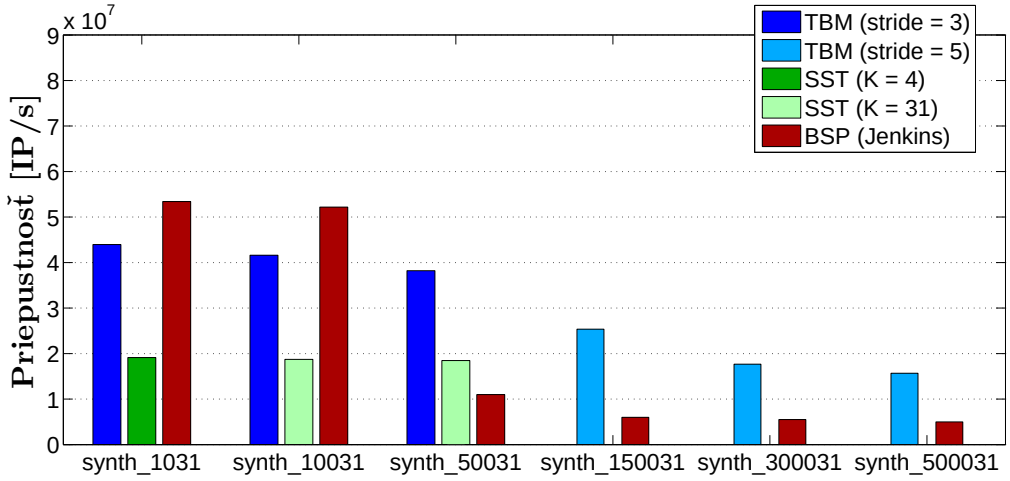
Obr. 5.9: Graf priepustnosti BSP na syntetických prefixových sadách

Na porovnanie priepustnosti medzi jednotlivými softvérovými algoritmami bol pre každý algoritmus vybraný jeho najefektívnejší variant na danej prefixovej sade. Algoritmus TBM bol najvýkonnejší väčšinou pre $stride = 3$ alebo $stride = 5$, len v jednom prípade mal najvyššiu priepustnosť pri $stride = 4$ (bgptable_161342). Algoritmus SST bol najvýkonnejší pre $K = 4$ alebo $K = 31$. Algoritmus BSP bol pre reálne sady s 1 009 a 53 781 prefixmi výkonnejší pri použití hašovania použitím horných bitov adries. Pre ostatné sady bolo efektívnejšie použitie Jenkinsnovej hašovacej funkcie. Výsledky BSP na syntetických sadách použitím výberu bitov IP adresy ako hašovacej funkcie neboli do výsledkov zarátané. Testované algoritmy dosahovali najlepšiu výkonnosť pri požití 8 vlákien.

Porovnanie algoritmov v najlepších konfiguráciách je zobrazené v grafoch na obrázkoch 5.10 a 5.11. Z grafov je možné vidieť, že na najmenších prefixových sadách má najlepšiu priepustnosť algoritmus BSP. Jeho výkonnosť však rýchlo klesá so stúpajúcou veľkosťou prefixovej sady a pre väčšie sady dosahuje lepšie výsledky algoritmus TBM s postupne rastúcou hodnotou použitej stride. Priepustnosť algoritmu SST je na každej testovanej prefixovej sade prekonaná minimálne jedným z ostatných algoritmov a SST pritom nedosahuje ani polovicu výkonnosti najlepšieho algoritmu.



Obr. 5.10: Graf porovnania priepustnosti SW algoritmov na reálnych sadách



Obr. 5.11: Graf porovnania priepustnosti SW algoritmov na syntetických sadách

5.2 Priepustnosť na embedded procesoroch

V spolupráci s vývojovou skupinou ANT@FIT boli vybrané z implementovaných algoritmov hľadania najdlhšieho zhodého prefixu testované na embedded procesoroch. Testované procesory sa využívajú v bežných smerovačoch a dokonca niektoré z testovacích platforiem boli smerovačmi. Testované boli algoritmy TBM pre $stride = 4$ (TBM4) a $stride = 5$ (TBM5) a SST pre $K = 7$ (SST7) a $K = 15$ (SST15). Všetky algoritmy boli testované na dvoch rôzne veľkých prefixových sadách, konkrétne bgptable_1009 a bgptable_10084.

Testovanie prebiehalo na platformách s rôznymi operačným systémami a rôznymi typmi procesorov. Zoznam platforiem je možné vidieť v tabuľke 5.2. K platformám sú uvedené detaily o procesoroch a použitom operačnom systéme. Nie všetky z platforiem podporovali viacvláknové spracovanie a knižnicu pthreads. Zdrojové kódy boli preto prepísané do tvaru bez použitia vlákien. Merania preto prebiehali len s použitím jedného vlákna. Konfiguračné súbory daných algoritmov a vstupné súbory boli vytvorené rovnakým spôsobom ako pri testoch na viacjadrovom počítači. Na testovanie boli vytvorené pre každú prefixovú sadu štyri vstupné súbory obsahujúce po 25 000, 50 000, 100 000 a 200 000 adries.

Platforma	Procesor	Operačný systém	Frekvencia [MHz]
Econa	Cavium Networks, CNS1102	Linux 2.6.16	250
Linksys WAG 160N	Broadcom, BCM6538	OpenWRT	300
D-Link DIR-825	Atheros, AR7161	OpenWRT	680
Avila GW2348	Intel, IXP425	OpenWRT	533
Seagate Dockstar	Marvell, Kirkwood	OpenWRT	1 200
Ubiquiti	Atheros, AR7161	OpenWRT	680

Tabuľka 5.2: Popis testovacích embedded platforiem

V tabuľke 5.3 sú uvedené výsledné namerané priepustnosti platforiem. Priepustnosť je uvádzaná v miliónoch IP adries za sekundu (M predstavuje mega, milión) a bola vypočítaná ako priemer daného algoritmu pre všetky 4 súbory vstupných adries pre danú prefixovú sadu. V tabuľke je ku každej platforme a prefixovej sade uvedené len označenie a priepustnosť najlepšieho z testovaných algoritmov. Z tabuľky jednoznačne vidno, že najlepším je algoritmus Tree Bitmap so *stride* = 4. Nameranou priepustnosťou sa mu na väčšine platforiem približovala verzia TBM so *stride* = 5, algoritmus SST dosahoval niekoľkonásobne horšie výsledky, podobne ako na pri testoch na viacjadrovom stroji.

Platforma	bgptable_1009		bgptable_10084	
	Algoritmus	Priepustnosť [MIP/s]	Algoritmus	Priepustnosť [MIP/s]
Econa	(netestované)		TBM4	0,087
Linksys WAG 160N	TBM4	0,905	TBM4	0,914
D-Link DIR-825	TBM4	2,349	TBM4	2,335
Avila GW2348	TBM4	1,467	TBM4	1,462
Seagate Dockstar	TBM4	0,474	TBM4	0,441
Ubiquiti	TBM4	2,401	TBM4	2,392

Tabuľka 5.3: Najlepšie priepustnosti embedded platforiem

5.3 Hardvérová architektúra

Syntéza výslednej hardvérovej architektúry prebehla programami XST a Precision na architektúru xc5vlx110. Spotreba zdrojov čipu syntetizovanej jednotky pre obe základné verzie je uvedená v tabuľke 5.4, spolu s taktujúcou frekvenciou a odhadovanou rádovou kapacitou. Uvedená kapacita bola odhadnutá na základe simulácie zaplnenia pamätí jednotky popísanej ďalej v tejto sekcii. Podrobnejšie informácie o spotrebe zdrojov aj pre iné konfigurácie je možné nájsť v prílohe C.

Z popisu jednotky v kapitole 4 je zrejmé, že operácia vyhľadania najdlhšieho zhodného prefixu je v nej vykonávaná v konštantnom čase, teda časová zložitosť vyhľadania je O(1). Celý proces vyhľadania jednej IP adresy trvá vždy 12 taktov, čomu pri použití maximálnej taktujúcej frekvencie z tabuľky (113 Mhz) zodpovedá latencia 106 ns. Jednotka je zároveň vďaka zreťazenému spracovaniu schopná spracovať v každom takte jednu IP adresu. Prie-

pustnosť jednotky je teda rovnaká ako taktujúca frekvencia a má hodnotu 113 000 000 IP adries za sekundu bez ohľadu na veľkosť použitej prefixovej sady.

Verzia	Slices	BRAMs	Frekvencia	Kapacita prefixov
FULL	1 940	54	113,25 MHz	8 500 IPv4 a 500 IPv6
LITE	1 661	30	113,25 MHz	2 000 IPv4 a 500 IPv6

Tabuľka 5.4: Parametre syntetizovanej HW jednotky

Veľkosť použiteľnej prefixovej sady je limitovaná kapacitou jednotky a teda ovplyvňuje len náročnosť jednotky na zdroje a nie priepustnosť. Pre podporu väčších prefixových sád je potrebné vytvoriť jednotku s viacerými atomickými hašovacími tabuľkami. Tabuľka 5.5 zobrazuje pamäťové nároky jednotky, potrebné pri reprezentácií rôzne veľkých prefixových sád. Teoretická veľkosť spotrebovanej pamäte uvedená v tabuľke je vypočítaná len z počtu hašovaných začiatkov prefixov, predpokladá teda dokonalé hašovanie so 100 % zaplnením tabuliek. Reálna veľkost spotrebovanej pamäte berie ohľad na nedokonalosť hašovacej funkcie a počíta s pevnými veľkosťami atomických hašovacích tabuliek. Na získanie počtu týchto tabuliek bol implementovaný program HWmem (HWmem.c), ktorý simuluje rozloženie prefixov hašovacou funkciou použitou v jednotke.

Počet použitých tabuliek závisí okrem veľkosti prefixovej sady aj od zvolenia vhodnej hodnoty semienka hašovacej funkcie. Program HWmem určil potrebné počty hašovacích tabuliek metódou Monte Carlo [18] ako minimum z 1 000 000 pokusov s náhodne generovanou hodnotou semienka. Získané hodnoty sú preto zaťažené chybou 0,1 % (vzťah $err = 1/\sqrt{N}$).

Prefixová sada	Hašovaných začiatkov	TBM uzly		Atomických haš tabuliek	Celková pamäť [KiB]	
		Krok 1	Krok 2		Teoretická	Reálna
bgptable_1009	1 002	457	339	6	17,96	58,13
bgptable_10084	9 495	3 931	3 331	21	156,12	249,94
bgptable_53781	40 084	10 368	17 286	62	577,44	760,29
bgptable_161342	103 300	14 241	39 952	135	1 264,64	1 537,61
bgptable_322684	193 500	15 844	57 072	233	2 094,69	2 446,97
synth_1031	1 002	3	3	6	8,89	49,06
synth_10031	10 002	3	3	19	79,21	153,06
synth_50031	50 002	3	3	67	391,71	537,06
synth_150031	150 002	3	3	176	1 172,96	1 409,06
synth_300031	300 002	3	3	334	2 344,83	2 673,06
synth_500031	500 002	3	3	539	3 907,33	4 313,06

Tabuľka 5.5: Pamäťové nároky HW jednotky

Hodnoty uvedené v stĺpci s celkovou spotrebou pamäte sú vypočítané pri dátovej šírke identifikátorov 32 bitov (4 Bajty). V implementovanej verzii architektúry sú používané len 16 bitové identifikátory, pretože na pokrytie implementovanej kapacity postačujú. Pri použití 32 bitových identifikátorov bude celková veľkosť jedného TBM uzla prvého kroku 16 B. Veľkosť TBM uzla druhého kroku (bez externej časti a uloženia LPM pre nezhodu) 6 B. Veľkosť jedného záznamu hašovacej tabuľky bude 8 B a veľkosť jednej hašovacej tabuľky potom 8 KiB. Nakoniec veľkosť tabuľky s priamym adresovaním, obsahujúcej 256 položiek

(ráta sa len jeden kontext) bude 1 KiB. Do celkovej potrebnej pamäte nie je zarátaná pamäť pre IPv6 blok, keďže prefixové sady neobsahujú žiadne IPv6 adresy.

5.4 Zhrnutie dosiahnutých výsledkov

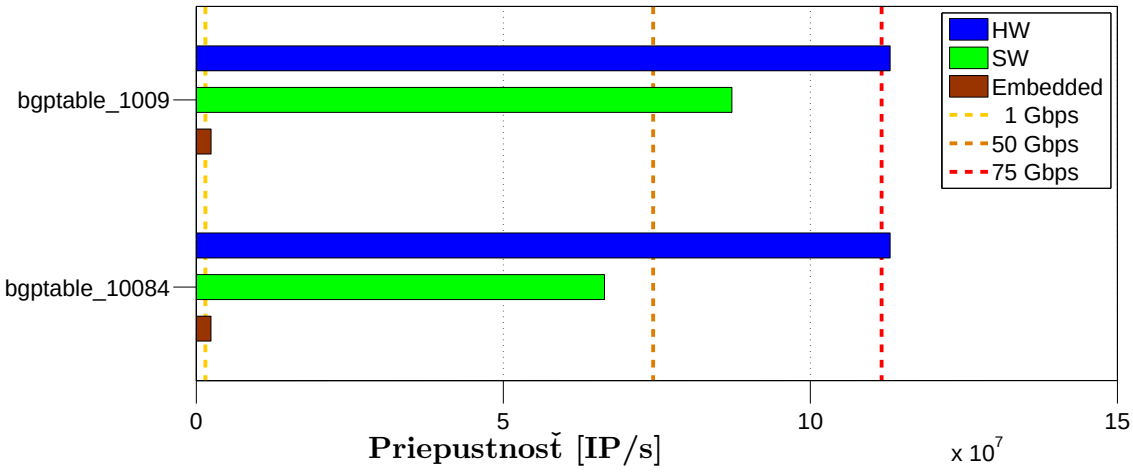
V rámci predošlých častí tejto kapitoly boli popísané testy merania a vyhodnotenia priepustnosti dosahovanej softvérovými algoritmami aj navrhnutým hardvérovým riešením. Výsledné priepustnosti boli doteraz vždy uvádzané v počte spracovaných IP adries za sekundu. V počítačových sieťach sa ale priepustnosť bežne meria v objeme prenesených dát za sekundu (Mbps, Gbps). Pre lepšiu predstavu o dosahovanej výkonnosti algoritmov je teda dobré previesť nameranú priepustnosť na tieto jednotky.

Prvý krokom prevodu je prechod od počtu IP adries za sekundu na počet paketov za sekundu. Každý paket obsahuje vždy len jednu cieľovú adresu a na základe nej je smerovaný sieťou. Priepustnosť vyhľadávania najdlhšieho zhodného prefixu v počte paketov je teda rovnaká ako v počte IP adries. Ďalej treba určiť dátovú šírku pásma, ktorá je potrebná pri prenose jedného paketu. Hodnota priepustnosti dát za sekundu sa udáva pre rámce na linkovej vrstve. Najrozšírenejšou formou realizácie linkovej vrstvy je v dnešnej dobe Ethernet [1]. Ethernet pracuje s rôzne veľkými rámcami, pričom v každom rámci je zabalený jeden paket. Pri prevode použijeme najhorší prípad, teda najmenšiu veľkosť rámcov. Každý Ethernet rámec sa skladá z troch častí – hlavičky (14 B), tela (46 B až 1 500 B) a päty (4 B). Minimálna veľkosť rámca je 64 B. Pred prenosom dát každého rámca je ešte prenášaná preambula (8 B) slúžiaca na synchronizáciu. Na vzájomné oddeľovanie rámcov sú do komunikácie vkladané medzi rámcové medzery dlhé štandardne 96 bitov (20 B). Celková dátová šírka potrebná na prenos jedného paketu (rámca) je preto minimálne 84 B. Maximálny počet paketov prenesených za sekundu na gigabitovej Ethernetovej linke je potom 1 488 095.

V grafe na obrázku 5.12 je zobrazené porovnanie výkonnosti jednotlivých riešení operácie hľadania najdlhšieho zhodného prefixu namerané v predošlých častiach kapitoly. Pre softvérové algoritmy testované na viacjadrovom stroji (SW) je zobrazený najvýkonnejší algoritmus. Pre testované embedded procesory je zobrazený najvýkonnejší algoritmus na najvýkonnejšom procesore. V grafe sú označené hranice priepustnosti potrebné na splnenie potrieb 1 Gbps, 50 Gbps a 75 Gbps linky na najkratších paketoch. Z grafu vidno, že najlepšiu priepustnosť má navrhnutá hardvérová architektúra (vyše 75 Gbps), o niečo horšie sú softvérové algoritmy na viacjadrovom stroji (okolo 50 Gbps) a výrazne zaostávajú priepustnosti dosahované na embedded procesoroch (len 1,6 Gbps). Uvedená priepustnosť hardvérovej architektúry je tak pri bežnom sieťovom prenose dostatočná aj na viac ako 100 Gbps linku.

Priepustnosti softvérových riešení dosahované pri reálnom použití v sieťach budú výrazne nižšie ako sú namerané hodnoty. K zníženiu dôjde hlavne z dôvodu nutnosti zdieľania procesorového času a ostatných prostriedkov počítača s ostatnými operáciami smerovania. Ďalší pokles priepustnosti je možné očakávať pri použití algoritmov na IPv6 adresy. Keďže sú IPv6 adresy 4-krát dlhšie oproti IPv4 adresám, môže dôjsť pri algoritmoch TBM a SST až k 4-násobnému poklesu výkonnosti. Pre algoritmus BSP nebude pokles až taký výrazný, ale stále bude viditeľný. Nižšiu výkonnosť ako je zakreslená v grafe 5.12 dosahujú softvérové riešenia aj pri použití na väčšie prefixové sady.

Popísané problémy vedúce k poklesu priepustnosti softvérových algoritmov v reálnych podmienkach nemajú vplyv na priepustnosť hardvérovej architektúry. Vďaka paralelizmu a zreťazení spracovania v hardvérových riešeniach, vedie potreba súbežného spracovania



Obr. 5.12: Porovnanie výkonnosti riešení

ďalších operácií smerovania len na zväčšenie využitej plochy čipu. Pridané operácie však priamo neobmedzujú výkonnosť jednotky hľadania najdlhšieho zhodného prefixu. V navrhnutom algoritme Hash Tree Bitmap je zároveň časová náročnosť operácie vyhľadania najdlhšieho zhodného prefixu nezávislá na dĺžke IP adresy, preto ani použitie IPv6 adries nevedie k zníženiu priepustnosti.

Veľkou výhodou navrhnutého hardvérového riešenia je aj nízka spotreba zdrojov a teda aj nízka cena. Spotrebou zdrojov (plochou čipu) je hardvérové riešenie na rovnakej úrovni s embedded procesormi. Ako je ale možné vidieť z výsledkov testov, embedded procesory dosahujú aj napriek tomu rádovo (cca. 50-krát) horšiu priepustnosť. Viacjadrové procesory sa síce výkonnostne viac priblížili hardvérovej jednotke, ale majú zároveň niekoľkonásobne väčšiu spotrebu zdrojov.

Kapitola 6

Záver

Táto bakalárska práca je zameraná na operáciu vyhľadávania najdlhšieho zhodného prefixu využívanú v sieťových zariadeniach najmä pri smerovaní a klasifikácii paketov. Hlavný dôraz bol kladený na priepustnosť navrhovaného riešenia s ohľadom na potreby dnešných aj budúcich vysoko rýchlostných počítačových sietí.

Prvým krokom bolo podrobné naštudovanie problematiky fungovania počítačových sietí založených na protokoloch IPv4 aj IPv6. Zameral som sa hlavne na mechanizmy súvisiace so smerovaním paketov a význam operácie vyhľadania najdlhšieho zhodného prefixu v tomto procese. Následne som naštudoval princíp fungovania niekoľkých algoritmov realizujúcich túto operáciu predstavených v odbornej literatúre. Z nich som potom vybral tri najvhodnejšie, ktoré som naštudoval podrobnejšie a rozobral v rámci práce. Sú to algoritmy Tree Bitmap a Shape-Shifting Trie založené na Trie a algoritmus Binary Seach on Prefixes využívajúci hašovacie tabuľky. Spomínané algoritmy som následne implementoval s využitím výhod viacvláknového spracovania na viacjadrových procesoroch.

Ďalším krokom bol návrh algoritmu a implementácia na ňom postavenej hardvérovej architektúry realizujúcej operáciu hľadania najdlhšieho zhodného prefixu. Pri návrhu boli ako základ použité myšlienky získané z naštudovaných algoritmov. Cieľom bolo odhaliť silné stránky jednotlivých realizácií a vhodne ich v návrhu spojiť. Výsledkom je algoritmus Hash Tree Bitmap, ktorý v sebe spája výhody Tree Bitmap a použitia hašovacích tabuliek podľa Binary Search on Prefixes. Výsledný algoritmus zároveň vhodne upravuje použité koncepty s využitím výhod hardvérovej implementácie. Na základe navrhnutého algoritmu som potom implementoval architektúru v jazyku VHDL a urobil syntézu na technológii FPGA.

Nakoniec som odmeral výkonnosť softvérových implementácií a hardvérovej architektúry. Zo softvérových algoritmov mali najlepšiu priepustnosť algoritmy Tree Bitmap a Binary Search on Prefixes. Tie dosiahli dostatočnú priepustnosť pre IPv4 adresy pri použití výkonného viacjadrového procesoru. Pri reálnom nasadení bude však dosahovaná priepustnosť nižšia, pretože súbežne s nimi budú vykonávané aj ďalšie procesy potrebné pri smerovaní. Zníženie priepustnosti je možné očakávať aj pri použití IPv6 adries. Nad rámec zadania boli následne algoritmy prerobené pre testovanie na embedded procesoroch bežne využívaných v sieťových zariadeniach. Namerané hodnoty priepustnosti na týchto procesoroch boli podstatne nižšie a zďaleka nedostačujúce pre potreby gigabitových sietí.

Hardvérová architektúra priepustnosťou prevýšila softvérové riešenia a dosiahla priepustnosť dostatočnú pre dnešné vysoko rýchlostné siete. Priepustnosť architektúry v najhoršom prípade na najkratších IPv4 paketoch je približne 75 Gbps. Pri nasadení v reálnej sieti postačuje výkonnostne architektúra aj na zvládnutie 100 Gbps linky. Priepustnosť

ostáva stála aj pri použití IPv6 protokolu a vďaka paralelizmu hardvérového riešenia nebude znížená ani potrebou súbežného vykonávania ďalších operácií smerovania. Výkonnosť architektúry je možné ešte zvýšiť prechodom z technológie FPGA na technológiu ASIC.

Veľkou výhodou hardvérovej jednotky je tiež nízky pomer spotreby zdrojov vzhľadom k dosiahnutému výkonu oproti softvérovým riešeniam. Výkonnostne jednotka niekoľkonásobne prekonáva algoritmy bežiace na viacjadrových procesoroch. Pri tom však spotrebou zdrojov a cenou je približne na rovnakej úrovni ako testované embedded procesory. Tento fakt dokazuje veľký význam a silu ukrytú v hardvérovo urýchľovaných riešeniach časovo náročných a kritických úloh.

Ďalším pokračovaním tejto práce by mohlo byť vylepšenie vlastností algoritmu Hash Tree Bitmap. Jednou z možností vylepšenia je zvýšenie percentuálneho zaplnenia hašovacích tabuliek. Dosiahnuť je to možné napríklad použitím CAM pamäte na uloženie príliš kolíznych prefixov alebo použitím rôznych hašovacích funkcií pre tabuľky. Nevyhnutné pre možnosť ďalšieho pokračovania vo vývoji je vytvorenie platformy na rýchlu implementáciu a jednoduché testovanie navrhovaných vylepšení. Nad rámec zadania som preto už spravil tento úvodný krok a implementoval v práci popísanú podobu algoritmu Hash Tree Bitmap v jazyku Python v rámci prostredia Netbench.

Literatúra

[1] Cisco Networking Academy. *CCNA Exploration Course Booklet: Network Fundamentals, Version 4.0.* Indianapolis, USA: Cisco Press, September 2009. Course Booklets. ISBN 978-1-58713-243-8.

[2] Matoušek, P. *Sít'ové aplikace a správa sítí: Architektura sítí, adresování, testování.* Brno: FIT VUT v Brně, 2010.

[3] Postel, J. *Internet Protocol* [RFC 791]. September 1981. Dostupné na URL: <http://www.ietf.org/rfc/rfc791.txt>.

[4] Deering, S. a Hinden, R. *Internet Protocol, Version 6 (IPv6) Specification* [RFC 2460]. December 1998. Dostupné na URL: <http://www.ietf.org/rfc/rfc2460.txt>.

[5] Hinden, R. a Deering, S. *IP Version 6 Addressing Architecture* [RFC 4291]. February 2006. Dostupné na URL: <http://www.ietf.org/rfc/rfc4291.txt>.

[6] Cisco Networking Academy. *CCNA Exploration Course Booklet: Routing Protocols and Concepts, Version 4.0.* Indianapolis, USA: Cisco Press, September 2009. Course Booklets. ISBN 978-1-58713-251-3.

[7] Eatherton, W. N. *Hardware-based Internet Protocol Prefix Lookups.* Saint Louis, Missouri: Washington University, 1999. S. 28–29. Diplomová práce. Dostupné na URL: <http://www.arl.wustl.edu/~jst/studentTheses/wEatherton-1999.pdf>.

[8] Srinivasan, V. a Varghese, G. Faster IP Lookups Using Controlled Prefix Expansion. *SIGMETRICS Performance Evaluation Review.* June 1998, roč. 26. S. 1–10. Dostupné na URL: <http://doi.acm.org/10.1145/277858.277863>. ISSN 0163-5999.

[9] Eatherton, W., Varghese, G. a Dittia, Z. Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *SIGCOMM Computer Communication Review.* April 2004, roč. 34, č. 2. S. 97–122. Dostupné na URL: <http://portal.acm.org/citation.cfm?doid=997150.997160>. ISSN 0146-4833.

[10] Song, H., Turner, J. a Lockwood, J. Shape Shifting Tries for Faster IP Route Lookup. In *Proceedings of the 13TH IEEE International Conference on Network Protocols.* Washington, USA: IEEE Computer Society, 2005. S. 358–367. Dostupné na URL: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1544635>. ISBN 0-7695-2437-0.

[11] Kim, K. S. a Sahni, S. IP Lookup by Binary Search on Prefix Length. In *Proceedings of the Eighth IEEE International Symposium on Computers and Communication*. Washington, USA: IEEE Computer Society, 2003. S. 77–82. Dostupné na URL: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1214104>. ISSN 1530-1346.

[12] Honzík, J. M. *Algoritmy IAL*. Brno: FIT VUT v Brně, leden 2011. S. 130–136. Studijní opora, verze 11-C.

[13] ANT@FIT. *Netbench - Accelerated Network Technologies* [online]. Updated 12 January 2011 [cit. 21. marec 2011]. Dostupné na URL: <http://merlin.fit.vutbr.cz/ant/netbench/index.html>.

[14] Egi, N., Greenhalgh, A., Handley, M. et al. Forwarding Path Architectures for Multicore Software Routers. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*. New York, USA: ACM, 2010. S. 3:1–3:6. PRESTO '10, č. 3. Dostupné na URL: <http://doi.acm.org/10.1145/1921151.1921155>. ISBN 978-1-4503-0467-2.

[15] Jenkins, B. Algorithm Alley: Hash Functions. *Dr. Dobb's Journal*. 1997, roč. 22, č. 9. S. 107–109, 115–116. ISSN 1044-789X.

[16] Skačan, M. *Algoritmy pro vyhledání nejdelšího shodného prefixu*. Brno: FIT VUT v Brně, 2010. S. 14–16. Bakalářská práce. Dostupné na URL: <http://www.fit.vutbr.cz/study/DP/all?id=9640>.

[17] Intel Corporation. *Intel Xeon Processor X5450* [online]. Updated 21:22 6 April 2011 [cit. 10. apríl 2011]. Dostupné na URL: <http://ark.intel.com/Product.aspx?id=34446>.

[18] Peringer, P. *Modelování a simulace*. Brno: FIT VUT v Brně, listopad 2010. S. 114–116. Studijní opora, verze 2010-11-03.

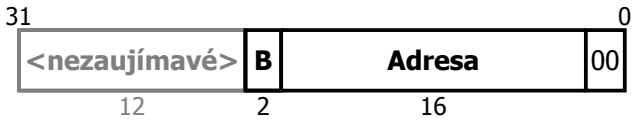
Dodatok A

Adresný priestor architektúry

Na prístup k adresnému priestoru architektúry je využívané štandardné pamäťové rozhranie MI32. Poskytuje zarovnaný zápis 32 bitových hodnôt a adresovanie 32 bitovými adresami po bajtoch. Pre správny zápis 32 bitových slov je preto potrebné používať adresy, ktorých hodnota je násobkom 4 (spodné 2 bity sú nulové). Hlavnými využívanými súčasťami MI32 sú signály:

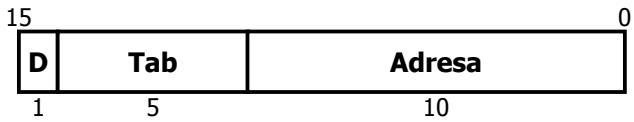
- **MI32.DWR** nesie 32 bitové dáta na zápis
- **MI32.ADDR** definuje 32 bitovú adresu pre zápis dát
- **MI32.WR** povoľuje zápis cez MI32 rozhranie

O rozdelenie adresného priestoru sa starajú adresné dekodéry na každej úrovni adresovania, ktoré správne distribuujú signál MI32.WR, povoľujúc tak zápis len na zvolenom mieste. Základné delenie priestoru medzi bloky architektúry je nasledujúce:



Kde B označuje blok, na ktorý sa adresovanie vzťahuje, *Adresa* je predaná adresnému dekodéru tohto bloku na ďalšie spracovanie, horných 12 bitov je ignorovaných a spodné 2 bity predstavujú zarovnanie. B môže nadobúdať hodnoty 00 pre IPv4 blok, 01 pre IPv6 blok a 10 pre TBM blok. Okrem toho sú špeciálne adresy priradené registrom uchovávajúcim semienko hašovacích funkcií. Ide o adresy s hodnotou $B = 11$ a *Adresou* nastavenou na číslo kontextu.

Adresný priestor IPv4 bloku je rozdelený nasledovne:

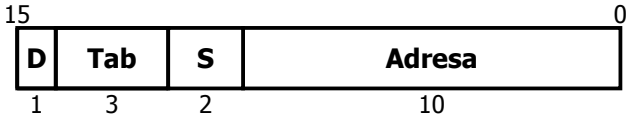


Kde Tab označuje číslo tabuľky so záznamami o začiatkoch prefixov, *Adresa* predstavuje samotnú adresu do vnútra zvolenej tabuľky a D vyberá typ dát v zázname. Bit D nastavený na 0 označuje zápis začiatku prefixu (kľúč záznamu) a 1 označuje zápis metadát záznamu. Dĺžku tabuliek je možné nastavovať zápisom na špeciálnu adresu, kedy $D = 1$, $Tab = 11111$ a *Adresa* je číslo tabuľky. Zapisované sú spodné 4 bity MI.DWR, vrchné dva platia

pre kontext 1 a spodné pre kontext 0. Kódovanie dĺžky je: 11 = 32 bitov, 10 = 24 bitov 01 = 16 bitov. Tabuľka s priamym prístupom nemá časť na ukladanie začiatkov prefixov ani možnosť nastavovať dĺžku (implicitne 00 = 8 bitov). Tvar metadát každého záznamu je:

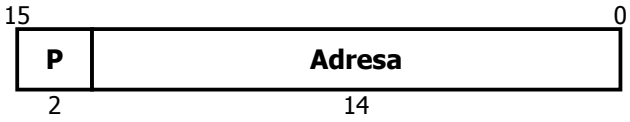
Zakreslených je 16 spodných bitov MI32.DWR, pri zápise metadát sa zvyšné nepoužívajú. Bity $K0$ a $K1$ reprezentujú platnosť záznamu v kontexte 0 resp. 1, bit Z predstavuje značku pokračovania začiatku prefixu v zázname (0 znamená so značkou, teda pokračuje). Nakoniec ID je identifikátorom prefixu alebo identifikátorom (adresou) dokončenia prefixu v TBM bloku.

Adresný priestor IPv6 bloku je rozdelený nasledovne:

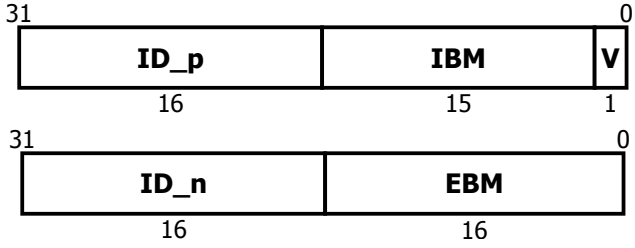


Význam položiek je rovnaký ako pre IPv4 blok a aj metadáta v zázname sú ukladané v rovnakom tvare. Naviac ale pribudol úsek adresy S označujúci 32 bitové dátové slovo v rámci jedného začiatku prefixu IPv6 adresy. Keďže rozhranie MI32 je 32 bitové a IPv6 adresa 128 bitová, je nutné nahrávanie začiatku IPv6 prefixu rozdeliť na štyri 32 bitové slová. Nastavovanie dĺžky tabuliek je podobné ako pri IPv4 bloku, teda $D = 1$, $Tab = 111$ a $Adresa$ je číslo tabuľky. Zapisovaných je spodných 8 bitov MI.DWR, vrchné štyri platia pre kontext 1 a spodné pre kontext 0. Kódovanie dĺžky je: 1111 = 128 bitov, 1110 = 120 bitov 1101 = 112 bitov, ...

Adresný priestor TBM bloku je rozdelený nasledovne:

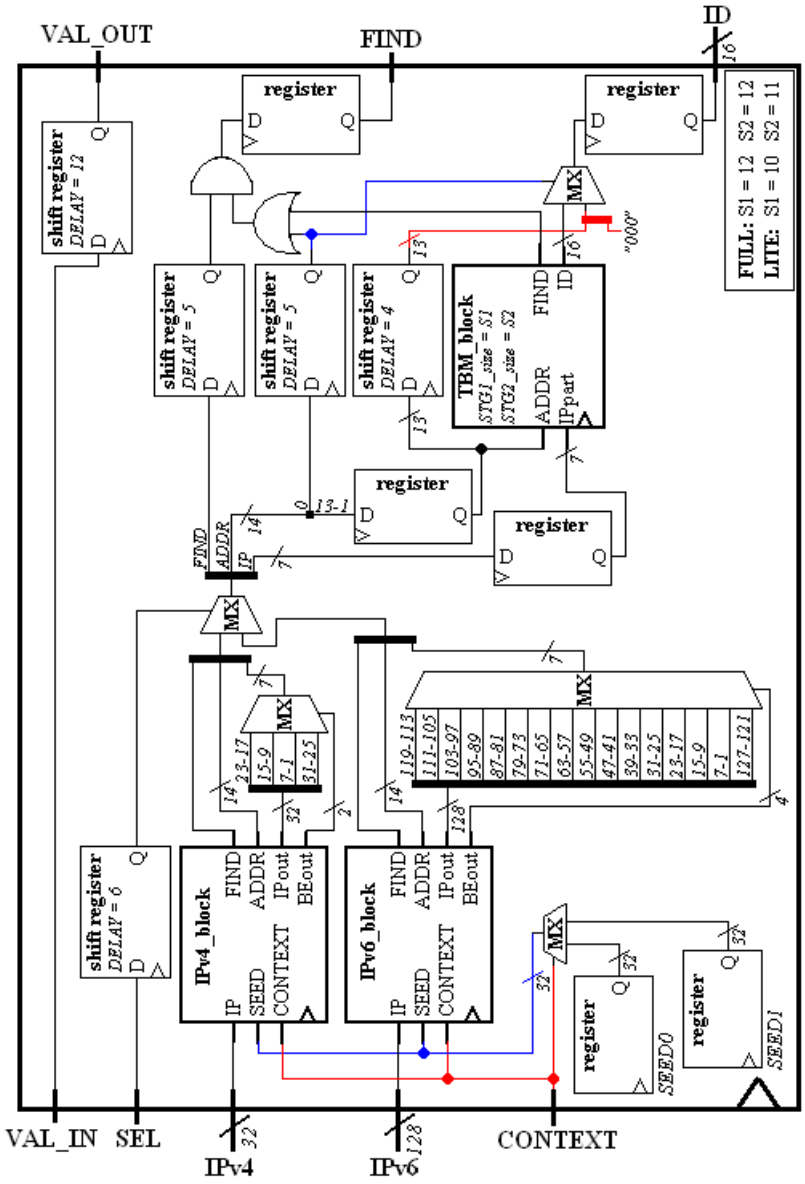


Kde P označuje pamäť na zápis a $Adresa$ predstavuje adresu do nej (identifikátor uzla). Hodnoty P sú 00 pre interné dáta uzla prvého kroku (identifikátor prvého prefixu – ID_p, interná bitmapa – IBM a platnosť uzla – V), 01 pre externé dáta uzla prvého kroku (identifikátor prvého následníka – ID_n a externá bitmapa – EBM), 10 pre interné dáta uzla druhého kroku a 11 pre dopredu vypočítaný identifikátor prefixu použitý v prípade nenájdenia zhodného dokončenia v TBM strome. Tvar dát internej a externej časti uzla je nasledujúci:

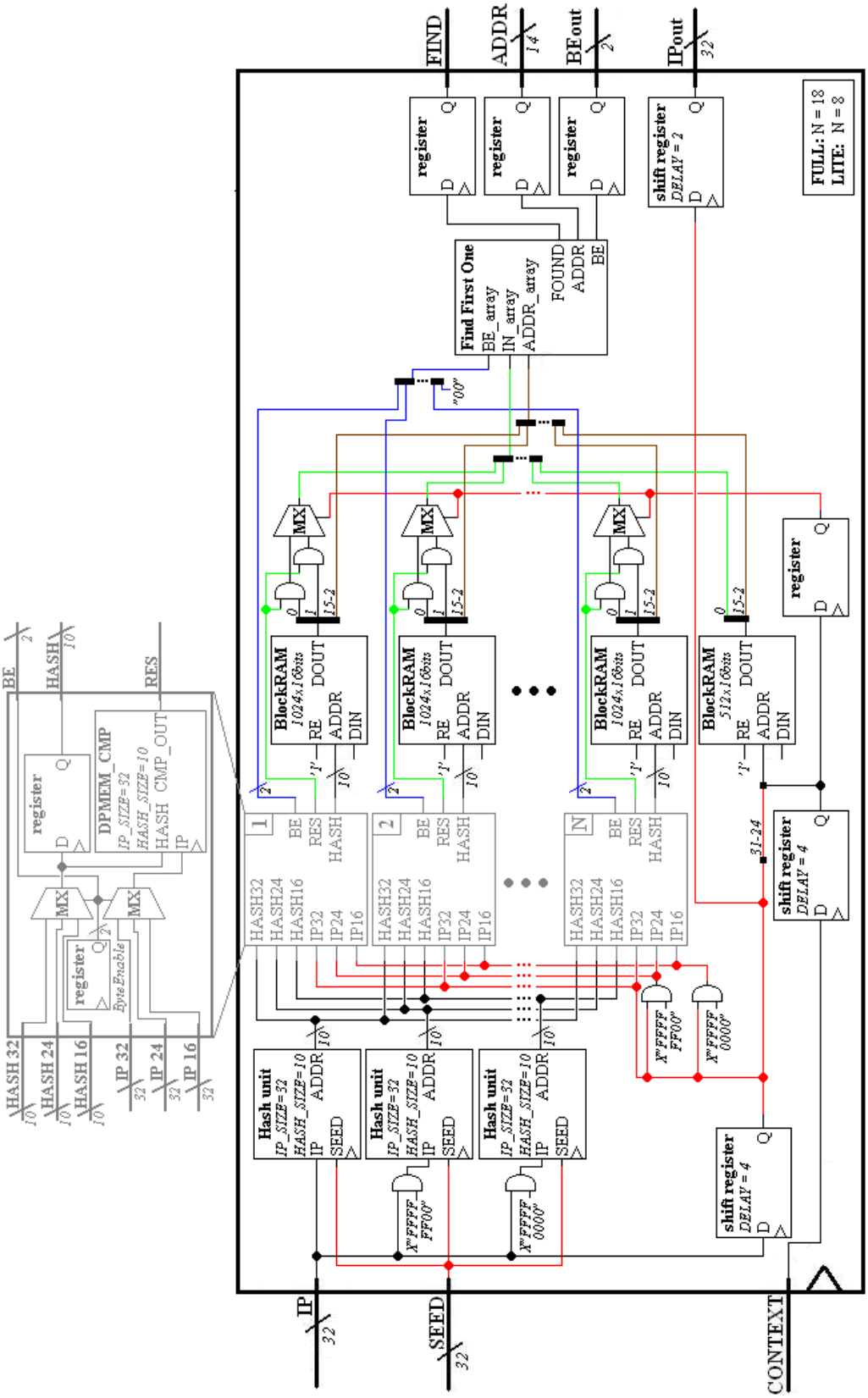


Dodatok B

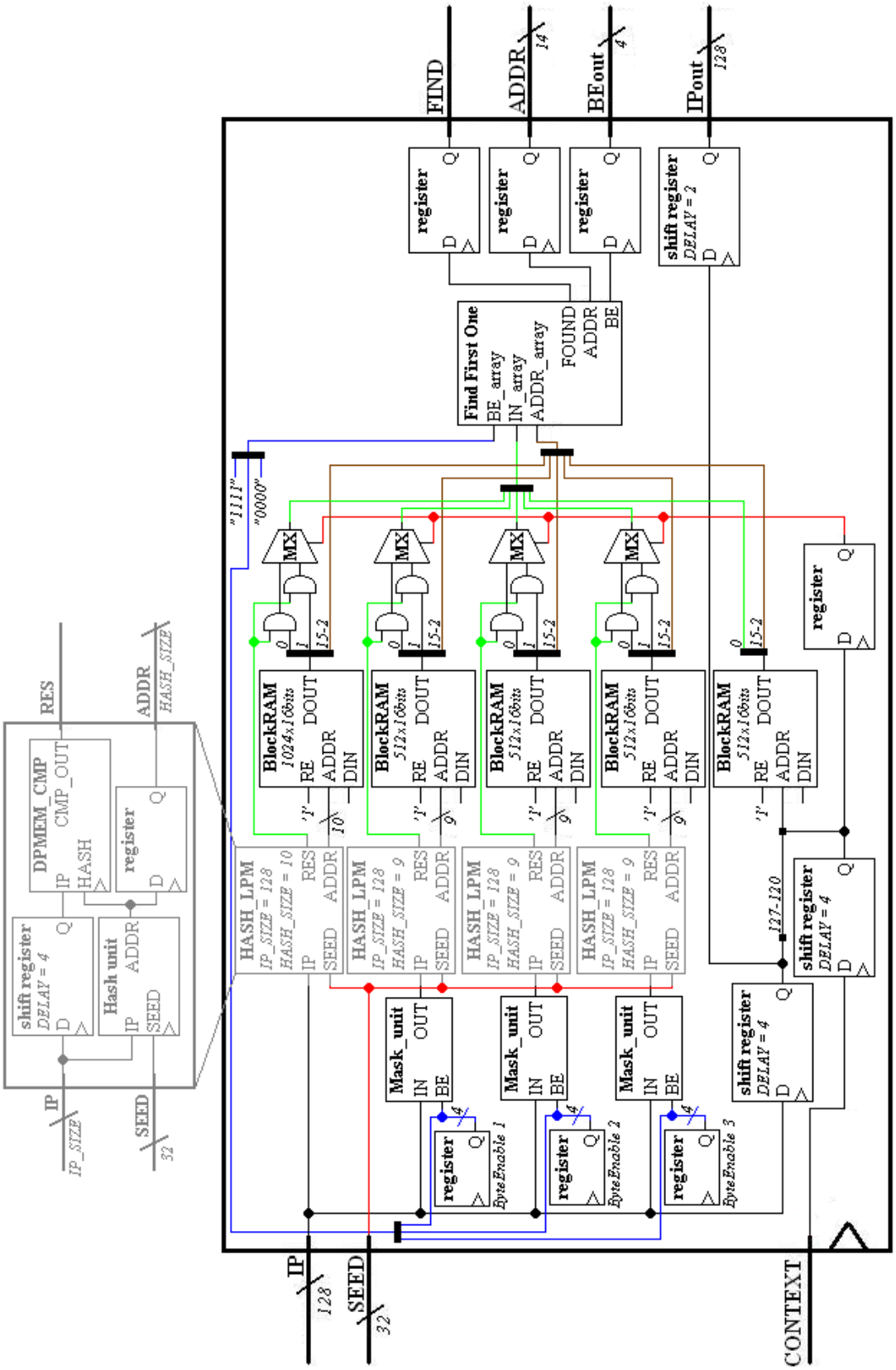
Podrobné schémy architektúry



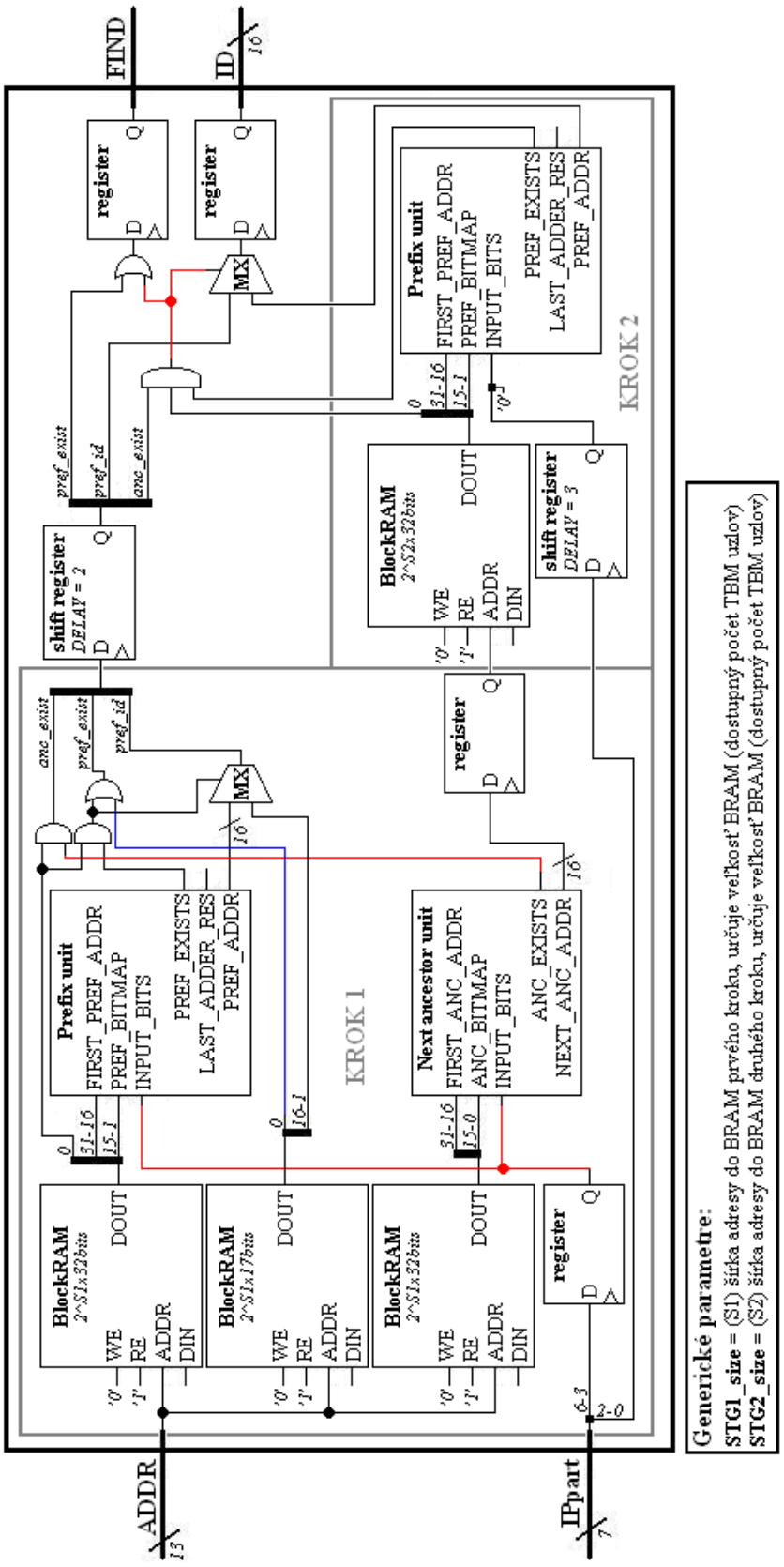
Obr. B.1: Podrobná schéma hardvérovej architektúry



Obr. B.2: Podrobná schéma IPv4 bloku (začiatky IPv4 prefixov)



Obr. B.3: Podrobná schéma IPv6 bloku (začiatky IPv6 prefixov)



Obr. B.4: Podrobná schéma TBM bloku (dokončenia prefixov)

Dodatok C

Spotreba zdrojov architektúry

Pri syntéze hardvérovej architektúry je možné nastavovať 4 základné parametre – počet hašovacích tabuliek IPv4 bloku (T_4), počet hašovacích tabuliek IPv6 bloku (T_6) a počet podporovaných uzlov TBM bloku v prvom (K_1) a druhom kroku (K_2). Hodnota každého z parametrov ovplyvňuje náročnosť architektúry na zdroje čipu, pritom ale každý parameter ovplyvňuje len jeden z blokov architektúry. Spotreba zdrojov bola preto zisťovaná po jednotlivých blokoch. Do tabuliek boli potom zapísané údaje o počte použitých BRAM blokov (BRAM) a CLB Slices (CLB) na čipe.

Závislosti zabratých zdrojov čipu od nastavených parametrov pre jednotlivé bloky sú uvedené v tabuľkách C.1 (IPv4 blok), C.2 (IPv6 blok) a C.3 (TBM blok). Uvedené hodnoty boli získané syntézou daného bloku s uvedenými hodnotami parametrov. Na konci každej tabuľky je naviac uvedený interpolačný vzťah na výpočet približnej spotreby zdrojov daného bloku. Vzťah bol vytvorený na základe zistených hodnôt uvedených v príslušnej tabuľke.

Na výpočet celkovej spotreby zdrojov architektúrou je nutné pre dané nastavenia sčítať príslušné spotreby jednotlivých blokov. Hodnoty sú uvedené priamo v tabuľkách alebo je možné ich dopočítať pomocou uvedených vzorcov. K takto získanej spotrebe je potrebné ešte pripočítať spotrebu hlavnej časti architektúry. Hlavnú časť neovplyvňujú nastaviteľné parametre, má preto konštantnú spotrebu zdrojov a to 27 CLB a 0 BRAM.

Hašovacích tabuliek	**SLC**	**BRAM**
4	532	7
6	572	10
8	620	13
10	668	16
14	766	22
18	864	28
22	960	34
26	1 057	40
30	1 155	46
T_4	$24,149 * T_4 + 428,7$	$1,5 * T_4 + 0.5$

Tabuľka C.1: Spotreba zdrojov IPv4 blokom

Hašovacích tabuliek	**SLC**	**BRAM**
2	455	8
3	685	10
4	908	13
5	1 142	15
6	1 350	18
7	1 574	20
T_6	$223,54 * T_6 + 13,057$	$2,5 * T_6 + 2,5$

Tabuľka C.2: Spotreba zdrojov IPv6 blokom

			Uzlov kroku 2					
			512	**1024**	**2048**	**4096**	**8192**	K_2
Uzlov kroku 1	**512**	**SLC**	81	89	90	100	118	
		BRAM	2	3	4	6	10	
	1024	**SLC**	96	105	105	115	134	
		BRAM	3	4	5	7	11	
	2048	**SLC**	98	106	107	117	135	
		BRAM	6	6	7	9	13	
	4096	**SLC**	120	129	130	141	159	
		BRAM	11	11	12	14	18	
	8192	**SLC**	162	171	173	184	203	
		BRAM	21	21	22	24	28	
	K_1	**SLC**	$0,01005 * K_1 + 0,0049 * K_2 + 80$					
		BRAM	$(5 * K_1 + 2 * K_2)/2048$					

Tabuľka C.3: Spotreba zdrojov TBM blokom