Vysoce výkonné vyhledávací algoritmy pro Ethernet

Bc. Tomáš Guzma

Diplomová práce

2024

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Tomáš Guzma**
Osobní číslo:	**A22591**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Vysoce výkonné vyhledávací algoritmy pro Ethernet**
Téma práce anglicky:	**High Performance Lookup Algorithms for Ethernet**

Zásady pro vypracování

1. Zpracujte literární rešerši na dané téma.
2. Vyberte několik algoritmů pro rychlé Ethernetové vyhledávání vhodných pro použití v embedded aplikacích.
3. Vybrané algoritmy implementujte na zvoleném Ethernetovém RISC-V akcelerátoru.
4. Změřte výkonnost implementovaných algoritmů.
5. Zpracujte srovnání implementovaných algoritmů a zhodnoťte jejich výhody a nevýhody.

Forma zpracování diplomové práce: **tištěná/elektronická**
Jazyk zpracování: **Slovenština**

Seznam doporučené literatury:

1. KIRSCH, A., MITZENMACHER, M. *Less Hashing, Same Performance: Building a Better Bloom Filter*. In: Azar, Y., Erlebach, T. (eds) Algorithms – ESA 2006. ESA 2006. Lecture Notes in Computer Science, vol 4168. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11841036_42.
2. PATTERSON David, WATERMAN Andrew. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017, 200 s. ISBN: 978-0999249116.
3. SPURGEON, Charles E., ZIMMERMAN, Joann. *Ethernet: The Definitive Guide, 2nd Edition*. O'Reilly Media, 2014, 527 s. ISBN 9781449363000.
4. TANENBAUM Andrew, WETHERALL David. *Computer Networks (5th Edition)*. Pearson, 2010, 980 s. ISBN: 978--0132126953.
5. SHI, Xiao a Nathan BRONSON. *Open-sourcing F14 for faster, more memory-efficient hash tables*. Engineering at Meta [online]. 2019. Dostupné z: https://engineering.fb.com/2019/04/25/developer-tools/f14/. [citováno. 2023-11-02].
6. JUNGHWAN Kim a MYEONG-CHEOL *Ko. A Novel Prefix Cache with Two-Level Bloom Filters in IP Address Lookup*. Applied Sciences [online]. 2020. Dostupné z: https://www.mdpi.com/2076-3417/10/20/7198. [citováno 2023-11-02].

Vedoucí diplomové práce: **Ing. Jan Dolinay, Ph.D.**
Ústav automatizace a řídicí techniky

Datum zadání diplomové práce: **5. listopadu 2023**
Termín odevzdání diplomové práce: **13. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
DĚKAN
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 5. ledna 2024

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 10.05.2024

Tomáš Guzma v.r.
podpis studenta

ABSTRAKT

Táto diplomová práca sa venuje hľadaniu vhodných algoritmov pre vysoko výkonné vyhľadávanie v Ethernete na zvolenom RISC-V akcelerátore od firmy NXP Semiconductors. Výsledkom diplomovej práce bude program, ktorý bude slúžiť ako základ pre benchmarkovanie vysoko výkonných vyhľadávacích algoritmov pre Ethernet. Prínosom tejto práce je spomínaný program a odporúčanie vhodného algoritmu na použitie pre RISC-V Ethernet akcelerátor pre firmu NXP Semiconductors. Teoretická časť sa je zameraná na popis Ethernetu, testovaných algoritmov a hardvéru, na ktorom porovnanie prebieha. Praktická časť sa venuje implementácií benchmarkovacieho programu, ktorý porovnáva vysoko výkonné vyhľadávacie algoritmy pre Ethernet.

Kľúčové slová: Ethernet, RISC-V, akcelerátor, hash tabuľky, C, C++, Python

ABSTRACT

This master thesis is devoted to finding suitable algorithms for high performance Ethernet search on a selected RISC-V accelerator from NXP Semiconductors. The result of the master's thesis will be a program that will serve as a basis for benchmarking high-performance search algorithms for Ethernet. The contribution of this thesis is the already mentioned program and the recommendation of a suitable algorithm to be used for the RISC-V Ethernet accelerator for NXP Semiconductors. The theoretical part is focused on the description of Ethernet, the algorithms tested and the hardware on which the comparison is performed. The practical part is devoted to the implementation of a benchmarking program that compares high-performance lookup algorithms for Ethernet.

Keywords: Ethernet, RISC-V, accelerator, hash tables, C, C++, Python

Chcel by som sa poďakovať môjmu vedúcemu práce Ing. Janovi Dolinayovi, Ph.D. za vedenie diplomovej práce a Ing. Adamovi Viktorinovi, Ph.D. za radu ohľadom vyhodnotenia benchmarkov. Taktiež sa chcem poďakovať za pripomienky a rady kolegovi Ing. Ondřejovi Špačkovi.

Prehlasujem, že odovzdaná verzia diplomovej práce a verzia elektronická nahraná do IS/STAG sú totožné.

OBSAH

ÚVOD

V modernom svete sa neustále zvyšujú nároky na výkon počítačových systémov. Zvýšené nároky vedú nie len k zdokonaľovaniu hardvéru, ale aj softvéru a sieťových aplikácií. Pre plné využitie hardvéru je potrebné softvérové a sieťové aplikácie optimalizovať. Porovnávanie algoritmov a dátových štruktúr nám pomáha vybrať najvýkonnejšie voľne dostupné algoritmy pre konkrétne, špecifické využitie.

Výkon je možné zvýšiť ako pomocou hardvérových, tak aj pomocou softvérových prostriedkov. Hlavným komponentom pri znižovaní latencie komunikácie v sieti Ethernet je využitie rýchleho algoritmu na vyhľadávanie, pretože v týchto zariadeniach sa vyhľadávanie vykonáva milióny krát za sekundu. Efektívny algoritmus musí byť predovšetkým rýchly a taktiež musí mať dobrú pamäťovú lokalitu aby sa minimalizovali drahé prístupy do operačnej pamäte. V neposlednom rade je pri embedded systémoch vždy kladený dôraz na minimálnu možnú spotrebu pamäte. V prípade NXP RISC-V akcelerátoru však prevažuje potreba pre rýchlosť potrebu úspory pamäte.

V rámci literárnej rešerše sa mi nepodarilo nájsť prácu s podobným zameraním. Avšak, existuje viacero porovnaní hash tabuliek, ktoré nie sú určené a spúšťané na embedded systémoch. Niektoré z týchto prác sú príliš staré a tým pádom nie sú relevantné.

Cieľom diplomovej práce je vytvorenie benchmarkovacieho programu pre porovnanie vysoko výkonných algoritmov pre použitie v Ethernet vyhľadávaní na embedded RISC-V akcelerátore od firmy NXP Secimoconductors. Po vytvorení bude tento program použitý na porovnanie spomínaných algoritmov. Finálnym výstupom práce bude odporúčanie algoritmu, ktorý bol v porovnaní najvýkonnejší.

Teoretická časť diplomovej práce je zameraná na zoznámenie čitateľa s konceptami a technológiami používanými v praktickej časti diplomovej práce. Prvá kapitola popisuje Ethernet, jeho históriu a jeho využitie v embedded systémoch. Druhá kapitola sa zaoberá algoritmami a dátovými štruktúrami, ktoré sú vhodné pre vyhľadávanie v Ethernete. Posledná kapitola teoretickej časti je zameraná na popis akcelerátorov, offloadov a technológií využívaných v RISC-V akcelerátore od firmy NXP Semiconductors.

Praktickou časťou diplomovej práce je benchmarkovací program, ktorý porovnáva voľne dostupné algoritmy vhodné pre využitie v Ethernete. V praktickej časti tejto práce sú popísané technológie použité na vytvorenie spomínaného programu a požiadavky na jeho implementáciu. Ďalej sú popísané fázy vývoja, ktoré zahŕňajú analýzu voľne dostupných

algoritmov pre využitie v Ethernete na RISC-V akcelerátore od NXP Semiconductors. Po analýze nasleduje popis implementácie benchmarkovacieho programu, kde je popísaná implementácia, problémy ktoré pri nej vznikli a ich riešenia. V rámci tejto práce bola okrem benchmarkovacieho programu implementovaná aj hash mapa a Bloom filter. Popis implementácie týchto dátových štruktúr je taktiež zahrnutý v tejto kapitole. V poslednej kapitole diplomovej práce sú zhrnuté a popísané výsledky benchmarkov.

I. TEORETICKÁ ČÁST

1 ETHERNET

Pojem Ethernet sa vzťahuje na skupinu produktov lokálnej siete (LAN), na ktorú nadväzuje norma od Institute of Electrical and Electronics Engineers (IEEE) 802.3, ktorý definuje to, čo je bežne známe ako protokol CSMA/CD – Carrier Sense Multiple Access with Collision Detection. [1] V tejto kapitole bude prebratá história Ethernetu, úvod do Ethernetu a jeho IEEE noriem. Záver tejto kapitoly sa bude zaoberať Ethernetom v embedded systémoch.

1.1 História Ethernetu

Počiatky technológie, ktorú dnes poznáme pod názvom Ethernet, siahajú až do 22. mája roku 1973 kedy Bob Metcalfe napísal memorandum opisujúce sieťový systém Ethernet, ktorý bol vyvinutý na prepojenie pokročilých počítačových pracovných staníc, vďaka čomu bolo možné posielať údaje medzi týmito počítačovými stanicami. Táto technológia bola počiatočne vyvíjaná vo výskumnom stredisku Xerox Palo Alto Research Center, PARC, v Kalifornii. Hlavný zámer Ethernetu, ktorý bol revolučný na svoju dobu, bolo jeho využitie na umožnenie komunikácie medzi počítačmi v LAN sieťach. Spomínané memorandum sa zaoberalo zdokonalením predošlého experimentu v oblasti vytváraní sietí s názvom „The Aloha Network“. Toto zdokonalenie spočívalo v tom, že novo vyvinutý systém načúva aktivite na sieti predtým ako začne vysielať, podporuje pripojenie viacerých zariadení a dokáže odhaliť kolízie, ktoré vzniknú v rámci siete. Tento prístup k Ethernetovému kanálu bol nazvaný „Carrier Sense Multiple Access with Collision Detection“ alebo CSMA/CD. [2]

Bob Metcalfe tento systém pôvodne nazval „Alto Aloha Network“, ale v roku 1973 zmenil jeho názov na „Ethernet“, pretože chcel aby bolo z názvu jasné, že Ethernet dokáže podporovať akýkoľvek počítač a nie len Altos. [2]

Ethernet vychádza zo základu slova „ether“, ktoré popisuje hypotetickú substanciu, pomocou ktorej fyzika do začiatku 20. storočia vysvetľovala šírenie elektromagnetických vĺn. Pomenovanie Ethernet teda prirovnáva fyzické médium, ktoré prenáša bity k hypotetickej teórií šíreniu elektromagnetický vĺn. [2]

Prvá norma Ethernetu bola publikovaná v roku 1980 a dosahovala rýchlosť 10 Mb/s. Táto norma je známa pod názvom DIX, pretože bola publikovaná konzorciom dodávateľov. DIX bol neskôr použitý ako základ pre IEEE 802.3 Táto IEEE norma bola publikovaná v

roku 1985 pod názvom „IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications“. [2]

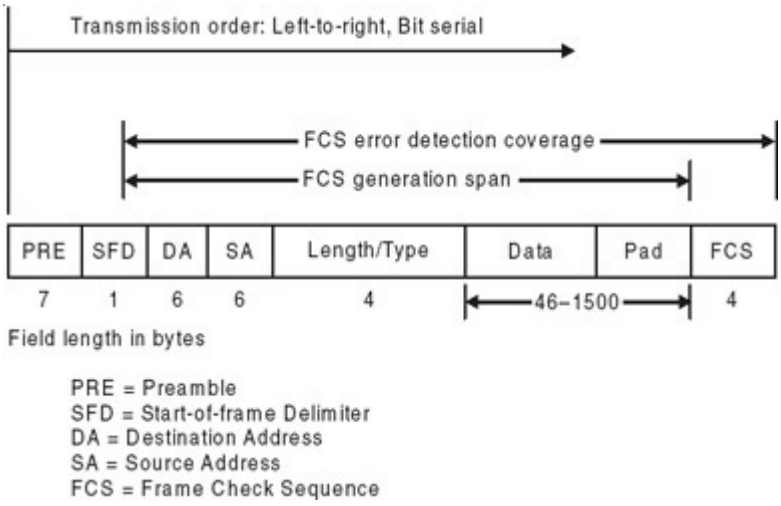
1.2 Popis Ethernetu

V dnešnej dobe je Ethernet najrozšírenejší spôsob pripojenia zariadení lokálnej siete k internetu. Spomínané zariadenia môžu byť osobné počítače, tlačiarne alebo embedded systémy. Tieto zariadenia je taktiež možné pomocou Ethernetu prepojiť vzájomne. Ethernet využíva na prenos dát fyzické médium, pomocou ktorého prepája zariadenia v sieti. Medzi populárne fyzické média patria koaxiálny kábel, krútená dvojlinka a optické vlákno.

Ethernet je popísaný v rôznych normách, ktoré popisujú jeho rýchlosť a použité fyzické médium. 10BASE-T je skrátený identifikátor ktorý určilo IEEE. Číslo 10 označuje maximálnu prenosovú rýchlosť 10 Mb/s. BASE sa vzťahuje na signalizáciu základného pásma, čo znamená, že môže prenášať iba signály Ethernetu na médiu. T znamená „twisted“ (krútené) ako v prípade krútenej dvojlinky. [1]

1.2.1 Ethernet rámce

Dáta odosielané cez Ethernet sú vo forme rámcov. Rámec Ethernetu obsahuje hlavičku, dátovú časť a pätičku. Záhlavie Ethernetu obsahuje adresy príjemcu aj odosielateľa. Existuje základný formát Ethernetového rámca, ktorý je špecifikovaný v norme IEEE 802.3.



Obrázok 1. Ethernet frame [1]

PRE (Preamble) pozostáva z postupnosti 7 bitov. Je to striedavý vzor jednotiek a núl, ktorý oznamuje prijímajúcim staniciam, že prichádza rámec. [1]

SOF (Start-of-frame delimiter) pozostáva z postupnosti 8 bitov. SOF je striedavý vzor jednotiek a núl, ktorý končí dvoma po sebe idúcimi bitmi 1, čo znamená, že nasledujúci bit je najľavejší bit v ľavom bajte cieľovej adresy. [1]

DA (Destination Address) sa skladá zo 6 bajtov a identifikuje, ktorá stanica by mala prijať rámec. Bit, ktorý je najviac naľavo v DA označuje, či je adresa individuálna alebo skupinová. Druhý bit zľava označuje, či je DA globálne spravovaná alebo lokálne spravovaná. Zvyšných 46 bitov je jednoznačne priradená hodnota, ktorá identifikuje jednu stanicu, definovanú skupinu staníc alebo všetky stanice v sieti. [1]

SA (Source Address) pozostáva zo 6 bajtov. Pole SA identifikuje odosielajúcu stanicu. SA je vždy individuálna adresa a najľavejší bit v SA je vždy 0. [1]

Lenght/Type pole sa skladá zo 4 bajtov. Toto pole udáva dĺžku a ID typu rámca ak je rámec zostavený pomocou voliteľného formátu. [1]

Data predstavuje postupnosť n bajtov s ľubovoľnou hodnotou, kde n je menšie alebo rovné maximálnej veľkosti rámca. Ak je dĺžka dátového poľa menšia ako 46, dátové pole sa musí rozšíriť pridaním výplne (Pad), ktorá je dostatočná na to, aby dĺžka dátového poľa dosiahla 46 bajtov.

FCS (Frame check sequence) obsahuje 32-bitovú hodnotu cyklickej redundancie, ktorú vytvára vysielajúca Media Access Control (MAC) a prepočítava prijímajúca MAC na kontrolu poškodených rámcov. FCS sa vytvára nad poľami DA, SA, dĺžka/typ a dátové polia. [1]

1.2.2 Rámcový prenos

Vždy keď koncová stanica MAC príjme požiadavku na prenos rámca so sprievodnými informáciami o adrese a údajoch z Logical Link Control podvrstvy, MAC začne prenosovú sekvenciu prenosom informácií do vyrovnávacej pamäte rámca MAC. Preamble a SOF oddeľovač sa vložia do polí PRE a SOF. Cieľová a zdrojová adresa sa vložia do polí adries. Dátové bajty sa spočítajú a počet bajtov sa vloží do poľa dĺžka/typ. Dátové bajty sa vložia do dátového poľa. Ak je počet dátových bajtov menší ako 46, pridá sa výplň, aby sa dĺžka dátového poľa zvýšila na 46 bajtov. Hodnota FCS sa generuje cez polia DA, SA, dĺžka/typ a dáta a pridáva sa na koniec dátového poľa. [1]

Po zostavení rámca bude jeho skutočný prenos závisieť od toho, či MAC pracuje v polo duplexnom alebo plne duplexnom režime. Norma IEEE 802.3 v súčasnosti vyžaduje, aby všetky Ethernetové MAC podporovali polo duplexnú prevádzku, v ktorej MAC môže vysielať alebo prijímať rámec, ale nemôže robiť oboje súčasne. Plno duplexná prevádzka je voliteľná schopnosť MAC, ktorá umožňuje MAC vysielať a prijímať rámce súčasne. [1]

V polo duplexnom Ethernete môže v danom čase prenášať údaje len jedna stanica. Stanica môže dáta iba vysielať alebo ich prijímať, ale nie oboje súčasne. [3]

Plno duplexný Ethernet umožňuje staniciam súčasne odosielať a prijímať údaje v sieti, čím sa eliminujú kolízie. To sa dosahuje pomocou plne duplexného prepínača LAN. Prepínanie siete Ethernet rozdeľuje veľkú sieť Ethernet na menšie segmenty. Plne duplexný Ethernet vyžaduje prenosové médium v podobe krútenej dvojlinky, karty sieťového rozhrania Ethernet a plne duplexný prepínač LAN. [3]

Príjem rámcov je v podstate rovnaký pri polo duplexnej aj plno duplexnej prevádzke s výnimkou toho, že plno duplexné MAC musia mať oddelené vyrovnávacie pamäte pre rámce a dátové cesty, aby umožnili súčasný prenos a príjem rámcov. [1]

Pri príjme rámca sa jeho cieľová adresa kontroluje a porovnáva so zoznamom adries, aby sa určilo, či je rámec určený pre túto stanicu. Ak sa nájde zhoda adresy, skontroluje sa dĺžka rámca a prijatá FCS sa porovná s FCS, ktorá bola vygenerovaná počas prijímania rámca. Ak je dĺžka rámca v poriadku a existuje zhoda FCS, typ rámca sa určí podľa obsahu poľa dĺžka/typ. Rámec sa potom analyzuje a odovzdá príslušnej vyššej vrstve. [1]

1.3 Ethernet v embedded systémoch

Embedded systém je špecializovaný počítačový systém, ktorý je zvyčajne integrovaný ako súčasť väčšieho systému. Pozostáva z kombinácie hardvérových a softvérových komponentov, ktoré tvoria výpočtový systém. Na rozdiel od stolových systémov, ktoré sú určené na vykonávanie všeobecnej funkcie, embedded systémy sú zvyčajne určené na vykonávanie špecifickej funkcie. [4]

Samotný Ethernet v embedded systémoch sa nelíši v porovnaním s Ethernetom použitým v osobných počítačoch. Niektoré embedded systémy dokonca využívajú kompletne rovnaký prístup aký je využitý v LAN sieťach pri osobných počítačoch. Avšak, niektoré embedded systémy vyžadujú, aby komunikácia cez Ethernet prebiehala v reálnom

čase. V tejto časti práce bude popísaný práve Ethernet pre tieto systémy a IEEE normy popisujúce komunikáciu v reálnom čase.

1.3.1 Využitie Ethernetu v embedded systémoch

Ethernet je často základom komunikácie v moderných embedded systémoch. Jeho využitie je rozšírené naprieč viacerými oblasťami embedded systémov. Embedded Ethernet sa často využíva aj v automobilovom priemysle, kvôli vlastnostiam, akými sú napríklad Audio Video Bridging (AVB), Power over Ethernet (PoE) a časová synchronizácia pomocou PTP, ktoré ho odlišujú od ostatných alternatív.

Asistenčné systémy v automobilovom priemysle sa spoliehajú na získavanie údajov z kamier a iných snímačov v reálnom čase. Pre tieto účely sa využíva AVB spolu s ďalšími protokolmi, ktoré zaisťujú kontrolovanú latenciu prenosu dát a garantujú šírku pásma prenosu. Jedným z týchto protokolov je SRP, ktorý rezervuje zdroje potrebné k podpore plynulého prenosu dát a notifikuje ostatné elementy v sieti. [5]

PoE je schopnosť prenášať energiu do zariadenia prostredníctvom Ethernetového kábla. Tento priemyselný štandard využíva skutočnosť, že jeden Ethernetový kábel môže prenášať dáta aj napájanie. Pomocou jedného kábla je možné zariadenie napájať a zároveň tento kábel používať na komunikáciu s ostatými zariadeniami. Táto vlastnosť eliminuje potrebu samostatného napájacieho kábla pre každé zariadenie, čo uľahčuje inštaláciu a šetrí fyzické miesto. [5][6]

Niektoré aplikácie a algoritmy využívané v automobilovom priemysle vyžadujú synchronizáciu medzi viacerými zariadeniami, senzormi alebo procesormi. Štandard, ktorý popisuje synchronizáciu časovačov v časovo senzitívnych aplikáciách pomocou Ethernetu je IEEE 802.1AS. Tento štandard je adaptáciou Precision Time Protocol (PTP), ktorý je určený na využitie s Audio Video Bridgingom a Time sensitive networkingom. [5]

1.3.2 Time sensitive networking

Time senstivite networking (TSN) predstavuje súbor viacerých nástrojov, ktorý ponúka spoľahlivosť, determinizmus a časovú synchronizáciu pre časovo senzitívne embedded zariadenia a je kritický z hľadiska bezpečnosti komunikácie prostredníctvom Ethernetu. Normy vzťahujúce sa na TSN sú založené na predchádzajúcej práci vykonanej v rámci pracovnej skupiny IEEE 802.1 na IEEE Audio Video Bridging. [7]

AVB normy ponúkajú niekoľko funkcií. Jednou z nich je spoločné poňatie času, ktoré sa dá dosiahnuť prostredníctvom súboru protokolov na synchronizáciu hodín, ktoré umožňujú koncovým staniciam a prepínačom, nazývané mosty v kontexte IEEE, navzájom synchronizovať svoje miestne hodiny za predpokladu, že ide o uzly, ktoré si uvedomujú čas, t. j. systémy ktoré explicitne odkazujú na čas. V norme sa definuje Best Master Clock Algorithm na výber časového referenčného uzla a Generalized Precision Time Protocol na synchronizáciu hodín uzlov, ktoré im poskytujú hodnotu hodín referenčného uzla, tzv. Grand Master. Ďalšiu funkciu, ktorú ponúka AVB, je rezervácia šírky pásma pre triedy v reálnom čase. Tá sa získava prostredníctvom Stream Reservation Protokol, ktorý umožňuje rezerváciu zdrojov v rámci mostov na ceste medzi zdrojom a cieľom. [7]

TSN rozširuje AVB súborom nástrojov pozostávajúcich z protokolov, ktoré pridávajú niekoľko ďalších funkcií. Tieto funkcie môžu návrhári sietí kombinovať, aby získali vlastnosti potrebné na splnenie potrieb aplikácie embedded systému. Spomínané funkcie zahŕňajú viazanú nízku latenciu, časovanie a synchronizáciu, vysokú spoľahlivosť a správu zdrojov. Normy TSN sa zameriavajú na jeden alebo viacero z týchto štyroch cieľov návrhu. [7]

1.3.3 Precision Time Protocol

Precision Time Protocol (PTP) je štandardizovaný protokol IEEE/IEC definovaný v IEEE 1588 s viacerými verziami PTP. Prvá verzia je z roku 2002 a je špecifikovaná v IEEE 1588-2002. PTP verzia 2 (PTPv2) bola oznámená v IEEE 1588-2008. Táto verzia nie je spätne kompatibilná s verziou 1588-2002. V dnešnej dobe je k dispozícii aj verzia IEEE 1588-2019, známa ako PTPv2.1. Je kompatibilná s PTPv2 a pridáva niektoré ďalšie funkcie. Ako už bolo spomenuté, PTP je využívaný na synchronizáciu hodín v počítačovej sieti. Je presný na menej ako mikrosekundu a meria sa v nanosekundách. [8] [9]

PTP systém je distribuovaný sieťový systém pozostávajúci z kombinácie zariadení PTP a iných zariadení. Zariadenia PTP zahŕňajú bežné hodiny, hraničné hodiny, transparentné hodiny typu end-to-end, peer-to-peer transparentné hodiny a riadiace uzly. Zariadenia, ktoré nie sú zariadeniami PTP, zahŕňajú mosty, routere a iné infraštruktúrne zariadenia. Protokol je distribuovaný a špecifikuje ako sa hodiny reálneho času v systéme synchronizujú navzájom. Tieto hodiny sú usporiadané do hierarchie master-slave. Hodiny s najvyššou autoritou v PTP hierarchii sa nazývajú Grandmaster hodiny. Tieto hodiny určujú referenčný čas pre celý systém. Synchronizácia sa dosahuje výmenou PTP správ, pričom

podriadené jednotky používajú na synchronizáciu svojich hodín čas nadriadeného zariadenia v hierarchii. [9]

1.3.4 Stream Reservation Protocol

Stream Reservation Protocol (SRP) môže využívať až tri signalizačné protokoly, Multiple MAC Registration Protocol (MMRP), Multiple Virtual LAN (VLAN) Registration Protocol (MVRP) a Multiple Stream Registration Protocol (MSRP). Využívajú sa na vytvorenie rezervácie streamu v rámci premosťovanej (bridged) siete.

Protokol MMRP sa v rámci SRP voliteľne používa na riadenie šírenia registrácií hovoriacich v celej premosťovanej sieti. [10]

Protokol MSRP je signalizačný protokol, ktorý poskytuje koncovým staniciam možnosť rezervovať sieťové zdroje, ktoré zaručia prenos a príjem dátových streamov v sieti s požadovanou kvalitou služby. Tieto koncové stanice sa označujú ako hovorcovia a poslucháči. Ako hovorcovia sú označované zariadenia, ktoré vytvárajú dátové streamy. Ako poslucháči sú na druhú stranu označované zariadenia, ktoré vytvorené dátové streamy spotrebúvajú. [10]

Protokol MVRP používajú koncové stanice a mosty na deklarovanie príslušnosti k VLAN, do ktorej sa vysiela „Stream“. To umožňuje, aby sa priorita dátového rámca šírila pozdĺž cesty od hovoriaceho k poslucháčom v označených rámcoch. [10]

Hovorcovia deklarujú atribúty, ktoré definujú vlastnosti streamu, takže mosty majú potrebné informácie k dispozícii, keď potrebujú prideliť prostriedky pre stream. Poslucháči deklarujú atribúty, ktoré požadujú príjem tých istých streamov. Mosty na ceste od hovoriaceho k poslucháčovi spracúvajú, prípadne upravujú a potom tieto deklarácie atribútov MSRP prepošlú ďalej. Mosty spájajú atribúty hovoriacich a poslucháčov prostredníctvom StreamID prítomného v každom z týchto atribútov, čo má za následok zmeny v rozšírených službách filtrovania a prideľovanie vnútorných zdrojov, keď sa streamy sprístupnia. [10]

2 VYHĽADÁVACIE ALGORITMY PRE ETHERNET

Pri výbere správneho vyhľadávacieho algoritmu pre Ethernet existuje viacero použiteľných možností. Ethernet vyhľadávanie je proces hľadania a párovania adries počítačov a iných zariadení, ktoré využívajú Ethernet. Tieto adresy sú zvyčajne ukladané v smerovacej tabuľke, v ktorej sa následne tieto údaje vyhľadávajú.

V tejto kapitole budú popísané vyhľadávacie algoritmy a dátové štruktúry medzi ktoré spadajú hash tabuľky, zhoda najdlhšieho prefixu, TCAM a Bloom filtre. Každý z týchto spôsobov so sebou nesie istú sadu výhod a nevýhod, ktoré sa musia pri výbere vhodného vyhľadávacieho algoritmu zvážiť. Táto kapitola taktiež upozorní na niektoré dôležité vlastnosti, ktoré definujú kvalitný vyhľadávací algoritmus pre Ethernet a metriky, ktoré sledujeme pri porovnávaní vyhľadávacích algoritmov.

2.1 Dôležité vlastnosti a metriky

Pri porovnaní vyhľadávacích algoritmov pre Ethernet existuje viacero dôležitých vlastností a metrík, ktoré môžu byť sledované a porovnávané.

V kontexte vyhľadávacích algoritmov pre Ethernet je samozrejme najdôležitejšou vlastnosťou rýchlosť. Môže ísť o rýchlosť vkladania, vyhľadávania alebo odstránenia zo smerovacej tabuľky. Operácie vkladania a odstránenia sú pri smerovacej tabuľke v Ethernet sieti menej podstatné, keďže frekvencia použitia týchto operácií je nižšia v porovnaní s operáciou vyhľadávania. Pri použití niektorých dátových štruktúr, akou je napríklad Bloom filter, je operácia odstránenia dokonca nemožná, alebo časovo náročná.

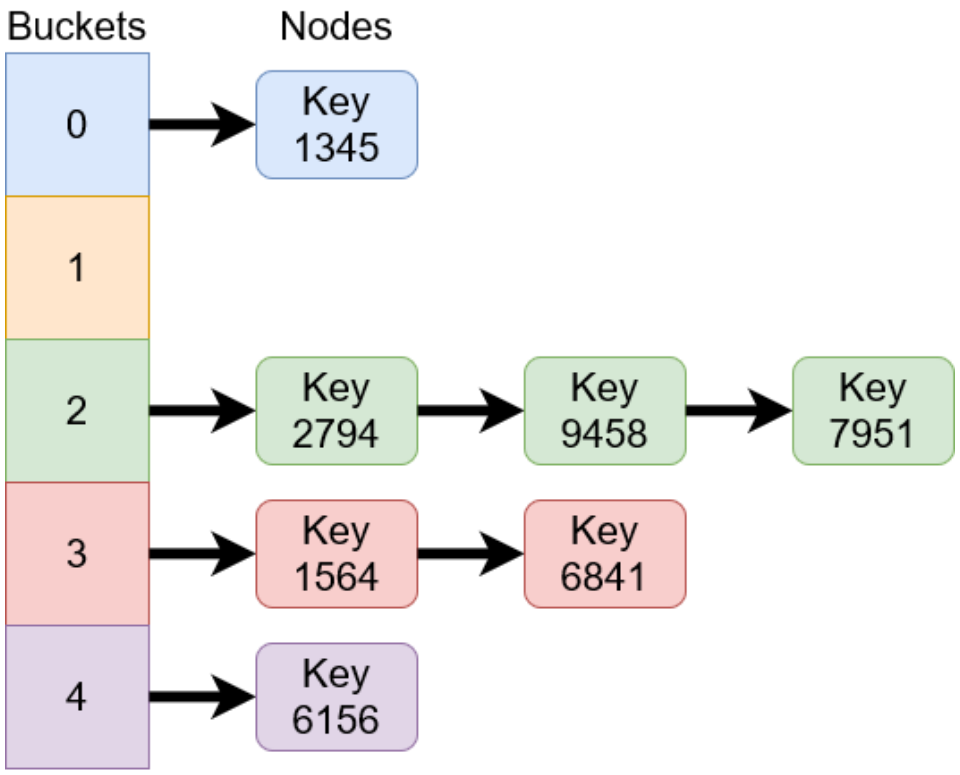
Ďalšia podstatná vlastnosť je pamäť, ktorú dáta zaberajú. Niektoré algoritmy a dátové štruktúry alokujú viac pamäte ako zaberajú samotné dáta. Hash tabuľky, ktoré používajú otvorené adresovanie, často alokujú až dvojnásobok potrebnej pamäte pre potencionálne zrýchlenie operácií vkladania, vyhľadania a odstránenia z hash tabuľky. Z týchto vlastností môžeme odvodiť mnoho metrík, pomocou ktorých môžeme tieto algoritmy a dátové štruktúry porovnávať. Pri výbere vhodného algoritmu pre Ethernetové vyhľadávanie v smerovacej tabuľke čelíme kompromisu medzi rýchlosťou a spotrebovanou pamäťou.

Lokalita referencie je podľa autora Williama Stallingsa „tendencia procesora pristupovať k rovnakej množine pamäťových miest opakovane počas krátkeho časového obdobia“. Jedným z typov lokality referencie sa považuje priestorová lokalita. Tento typ hovorí o tom, že procesor má tendenciu často pristupovať k prvkom, ktoré sú v pamäti umiestnené blízko práve spracovávaného prvku. Podtypom priestorovej lokality je

sekvenčná lokalita, kde procesor často pristupuje k prvkom, ktoré sú uložené v pamäti za sebou. Využitie lokality referencie vo vyhľadávacích algoritmoch pre Ethernet je jednou z vlastností, ktoré výrazne zvyšujú výkon daného algoritmu. Sekvenčnú lokalitu je možné jednoducho dosiahnuť tým, že dáta budú uložené v pamäti sekvenčne. Niektoré algoritmy a dátové štruktúry sú však postavené tak, že sekvenčnú lokalitu nedosahujú. Ako príklad je možné uviesť hash tabuľky so separátnym reťazením. Prvky nie sú v tejto tabuľke umiestnené vedľa seba, nakoľko sa tabuľka skladá z poľa ukazovateľov a nie samotných dát. [11]

2.2 Hash tabuľky

Hash tabuľky, ktoré sú taktiež známe pod pojmom hash mapy, predstavujú dátovú štruktúru, ktorá uchováva dáta v asociatívnych poliach. To znamená, že položky, ktoré sú vkladané do hash tabuľky, sa skladajú z párov kľúčov a hodnôt. Kľúče a hodnoty v hash tabuľke môžu byť ľubovoľného dátového typu a môžu sa líšiť, resp. kľúč nemusí byť rovnakého dátového typu ako hodnota. Bežná časová zložitosť hash tabuliek pre operácie vloženia páru, vyhľadania hodnoty a odstránenia páru je O(1) v najlepšom prípade a O(n) v najhoršom. Niektoré moderné hash tabuľky môžu dosahovať lepšiu časovú zložitosť v najhoršom prípade O(log n). [12]



Obrázok 2. Hash tabuľka s reťazením

Na obrázku 2. môžeme vidieť hash tabuľku, ktorá používa reťazenie. V tejto kapitole bude hlbšie popísaná a vysvetlená. Z obrázku je možné vidieť, že kľúč sa podľa hash funkcie namapuje na určitý index a následne sa pomocou reťazenia uloží na koniec lineárneho listu.

Princíp hash tabuliek spočíva v ukladaní dát do tabuľky pomocou hash funkcie. Táto hash funkcia je taktiež používaná pri vyhľadávaní hodnoty v tabuľke pomocou kľúča. Najjednoduchšou implementáciou hash tabuľky je zvyčajne pole lineárnych zoznamov. Každý element poľa sa v hash tabuľke nazýva „bucket“. V tomto prípade bucket predstavuje lineárny zoznam, v ktorom sú uložené jednotlivé elementy hash tabuľky. Pomocou tejto implementácie je najjednoduchšie predstaviť operácie vloženia, vyhľadávania a odstránenia z hash tabuľky. [12]

Vloženie do popísanej hash tabuľky spočíva v spočítaní hashu pomocou hash funkcie, vypočítanie indexu bucketu a vyhľadanie konca lineárneho zoznamu v danom buckete. Existuje mnoho hash funkcií, ktoré môžu byť pre tento účel použité. Tie najvýkonnejšie hash funkcie sa vyznačujú svojou rýchlosťou a rovnomerným rozložením svojho hashovania. Index bucketu do ktorého vkladáme pár je popísaný nasledujúcim vzorcom, kde *m* označuje množstvo alokovaných bucketov v pamäti. [12]

$$Index = Hash(Key)\ mod\ m$$

Rovnica 1. Výpočet indexu hash tabuľky

Vyhľadávanie v hash tabuľke je jednoduchá a rýchla operácia. Index bucketu sa vypočíta podobne ako pri vkladaní do hash tabuľky. Následne sa prechádza lineárny zoznam pokým sa hodnota kľúča v buckete nenájde alebo neprídeme na koniec lineárneho zoznamu. V najlepšom prípade je časová zložitosť O(1), pretože pokiaľ bucket obsahuje iba jeden element, tak vieme s istotou povedať, že práve tento element je alebo nie je v tabuľke bez toho, aby sme prehliadali iné elementy v hash tabuľke. Najhorší prípad O(n) by mohol nastať v prípade, že sú všetky elementy v hash tabuľke umiestnené práve v buckete, v ktorom hľadáme náš pár. Táto situácia je však veľmi nepravdepodobná ak používame kvalitnú hash funkciu. [12]

Pri odstránení páru z hash tabuľky sa najprv vykoná vyhľadávanie, ktoré nájde pár v buckete a následne sa tento pár môže odstrániť. Avšak, nemôžme zabudnúť na to, že na zachovanie integrity lineárneho zoznamu musíme pred odstránením tohto páru najskôr

naviazať predošlý pár na nasledujúci pár. Tento krok samozrejme môžeme preskočiť ak ide o posledný element v lineárnom zozname. [12]

$$koeficient\ zataženia = \frac{n}{m}$$

Rovnica 2. Výpočet koeficientu zaťaženia hash tabuľky

Jedným z podstatných parametrov hash tabuliek je koeficient zaťaženia. Koeficient zaťaženia určuje pomer vložených párov v hash tabuľke a počet alokovaných bucketov. Maximálny koeficient zaťaženia predstavuje reálnu hodnotu, ktorá určuje, že ak koeficient zaťaženia dosiahne alebo presiahne túto hodnotu, tak bude musieť byť hash tabuľka prehashovaná. Maximálny koeficient zaťaženia je fixná hodnota a koeficient zaťaženia rastie spolu s vloženými pármi v hash tabuľke. Nastavuje sa podľa potreby, ktorú hash tabuľka spĺňa. Príliš nízky maximálny koeficient zaťaženia plytvá pamäťou a naopak príliš vysoký maximálny koeficient zaťaženia spôsobuje degradáciu výkonu hash tabuľky. [13]

K prehashovaniu hash tabuľky dochádza vtedy, keď koeficient zaťaženia presiahne maximálny koeficient zaťaženia. Preplnenie hash tabuľky spôsobuje, že výkon hash tabuľky degraduje. V hash tabuľke uvedenej na Obrázku 2. môže nastať situácia, kedy sú lineárne zoznamy v jednotlivých bucketoch príliš dlhé a hash tabuľka degraduje v rýchlosti vyhľadávania, vkladania a odstraňovania. Pri prehashovaní hash tabuľky sa zvýši počet bucketov. Každý pár sa následne vloží na novo vypočítaný bucket v hash tabuľke. Na rozširovanie hash tabuliek sa zvyčajne používajú prvočísla alebo mocniny dvojky. [13]

2.2.1 Separátne reťazenie

Hash tabuľka, ktorá bola popísaná v časti vyššie, používa separátne reťazenie, ktoré sa niekedy taktiež nazýva „uzavreté adresovanie“. Separátne reťazenie predstavuje spôsob uchovávania párov v hash tabuľke, pri ktorom je použitá ďalšia dátová štruktúra na uchovanie hodnôt. Dátové štruktúry, ktoré sa zvyčajne využívajú na uchovanie hodnôt v hash tabuľke s reťazením, zahŕňajú spomenuté lineárne zoznamy, dynamické polia a samo vyvažujúce Binary Search Tree. [13]

Pri separátnom reťazení môže byť v jednom buckete umiestnený akýkoľvek počet párov. Z toho vyplýva, že koeficient zaťaženia môže presiahnuť hodnotu 1, keďže v hash tabuľke môže byť uložené väčšie množstvo hodnôt v porovnaní s počtom bucketov. Maximálny koeficient zaťaženia sa pri separátnom reťazení obvykle nastavuje v rozmedzí jedna až tri. Toto nastavenie drasticky ovplyvňuje výkon, keďže pri vyššom zaťažení je nutné prejsť cez väčšie množstvo položiek pri hľadaní hodnoty v hash tabuľke. [13]

2.2.2 Otvorené adresovanie

Na rozdiel od uzavretého adresovania, sa pri otvorenom adresovaní nepoužíva žiadna iná dátová štruktúra na ukladanie hodnôt. Hodnoty sú mapované v základnom poli a ukladané priamo v bucketoch. [13]

Obrázok 3. Hash tabuľka s otvoreným adresovaním

Na obrázku 3. môžeme vidieť zobrazenie ukladania dát do hash tabuľky s otvoreným adresovaním. Tento typ hash tabuľky bude popísaný v tejto kapitole. Z obrázku je možné vidieť, že kľúč sa podľa hash funkcie namapuje na určitý index, do ktorého sa vloží hodnota. Ak na jeho indexe už hodnota leží, posunie sa o jeden index ďalej. Táto technika sa nazýva lineárne próbovanie a taktiež bude popísaná nižšie.

Jedna z výhod otvoreného adresovania je lepšia lokalita pamäte. Keďže pri otvorenom adresovaní na ukladanie hodnôt využívame pole, tak je samozrejmé, že hodnoty budú v pamäti umiestnené za sebou a dosiahneme lepšiu pamäťovú lokalitu ako pri spomenutom lineárnom zozname. Pri otvorenom adresovaní taktiež nie je nutné alokovať pamäť pri každom vložení a uvoľňovať ju pri každom odstránení. Pri otvorenom adresovaní

sa alokuje určitý počet miest a po presiahnutí počtu daným koeficientom zaťaženia sa alokuje dodatočná pamäť. [13]

Pre vloženie hodnoty do hash tabuľky s otvoreným adresovaním sa po vypočítaní ideálneho indexu pomocou hash funkcie skontroluje, či je zodpovedajúci bucket voľný. Ak nie je voľný, tak nastane „kolízia“. Kolízie sa riešia rôznymi spôsobmi, ako napríklad lineárnym próbovaním. [13]

Lineárne próbovanie znamená, že ak mapovaný bucket nie je voľný, tak je skontrolovaný nasledujúci bucket v hash tabuľke. Tento proces sa opakuje, dokým element nie je nájdený. Ak je prehliadaný bucket voľný a element sa stále nenašiel, tak to znamená, že element sa v hash tabuľke nenachádza. [13]

Ďalší často využívaný druh riešenia kolízií je kvadratické próbovanie. Od lineárneho próbovania sa odlišuje iba tým, že každou iteráciou sa vytvárajú kvadraticky väčšie skoky medzi prehliadanými elementami. Tento spôsob próbovania zabraňuje primárnemu „clusteringu“. Clustering je jav, kedy sú elementy v tabuľke zhluknuté v skupinách a tým sa znižuje rýchlosť hash tabuľky. [13]

Maximálny koeficient zaťaženia je možné nastaviť v rozsahu od nuly po jedna. Na rozdiel od separátneho reťazenia, koeficient zaťaženia nie je možné nastaviť na hodnotu vyššiu ako jedna. Ideálne nastavenie tejto hodnoty sa väčšinou odvíja od použitej próbovacej techniky a od požiadaviek na výkon hash tabuľky. [13]

2.2.3 Hashovacie techniky

V tejto časti budú popísané hashovacie techniky, ktoré hash tabuľky implementujú ako riešenie kolízií pri vkladaní elementu. Všetky techniky, ktoré sú popísané v tejto sekcii využívajú otvorené adresovanie.

Robin Hood hashovanie rieši kolízie tým, že pri vkladaní elementu „berie bohatým a dáva chudobným“. Pri tomto hashovaní pracujeme s hodnotou Probe Sequence Lenght (PSL), ktorá predstavuje o koľko miest leží element od svojho ideálneho indexu. Elementy, ktoré od svojho ideálneho indexu ležia ďalej, majú vysoké PSL a sú označované ako chudobné. Element ktorý leží na svojej ideálnej pozícií má hodnotu PSL rovnú nule a je označovaný ako bohatý. Pri vkladaní elementu je jeho hodnota PSL nastavená na nulu. Ak je zahashovaní index vkladaného elementu voľný, element sa vloží na toto prázdne miesto. V opačnom prípade sa porovná hodnota PSL vkladaného elementu a elementu, ktorý leží na tomto indexe. Ak je hodnota PSL vkladaného elementu vyššia ako hodnota elementu, ktorý leží na danom indexe, tak sa vkladaný element vloží na miesto pôvodného elementu

a podobným spôsobom sa ďalej hľadá voľné miesto pre tento element. Toto sa opakuje, pokým sa nenájde prázdne miesto pre element. Vyhľadávanie je v najlepšom prípade O(1) a O(ln n) v najhoršom prípade [12]

Hopscotch hashovanie vytvára v hash tabuľke susedstvá o veľkosti H. Každý bucket v hash tabuľke je súčasťou H susedstiev. Pri vyhľadávaní elementu v tabuľke, ktorá používa hopscotch hashovanie, sa prehľadáva iba susedstvo, do ktorého element patrí. Konkrétne susedstvo je vybrané na základe indexu, na ktorý sa element nahashuje. Zložitosť vyhľadávania je v najhoršom prípade konštantná O(1), keďže vyhľadávanie prechádza najviac H bucketov. Pri vkladaní elementu sa hash tabuľka snaží vložiť element vždy maximálne H miest od ideálneho indexu. V tabuľke sa posúvajú elementy tak, aby sa uvoľnilo miesto v danom susedstve. Ak takýto posun nie je možný, tak je tabuľka rehashovaná a rozšírená na väčšiu kapacitu. [12]

Cuckoo hashovanie rieši kolízie v hash tabuľke implementáciou dvoch hash tabuliek. Pri cuckoo hashovaní sa vytvára sekundárna hash tabuľka. Veľkosť obidvoch tabuliek je rovná polovici normálnej veľkosti, teda N/2. Kolízie sa riešia presunutím existujúcich kľúčov z jednej hash tabuľky do druhej. Tento prístup je pomenovaný po spôsobe, akým kukučie mláďa vytláča vajíčko z hniezda, aby si uvoľnilo miesto. Ak je pri vkladaní elementu na jeho ideálnom mieste element, tak je vkladaný element vložený na toto miesto a pôvodný element presunutý do sekundárnej tabuľky. Ak je v sekundárnej tabuľke miesto taktiež obsadené, tak sa proces opakuje a pôvodný element tohto miesta hľadá opäť miesto v primárnej hash tabuľke. Tento prístup zaisťuje konštantný čas vyhľadávania O(1), keďže stačí skontrolovať jedno miesto v obidvoch tabuľkách. Keď nastane situácia, že elementy sa presúvajú a nedokážu nájsť voľné miesto, tabuľka je rehashovaná a rozšírená na väčšiu kapacitu. [12]

2.2.4 Hash funkcie

Pre výkon hash tabuľky je hash funkcia najpodstatnejšia operácia. Ide o matematickú funkciu, ktorá s pomocou kľúča priradí páru index v hash tabuľke. Neexistuje univerzálne najlepšia hash funkcia pre všetky typy a veľkosti hash tabuliek. Použitím hash funkcie na konkrétny kľúč, dostávame vždy rovnaký hash. Pomocou modulo operácie následne získavame index elementu umiestneného v hash tabuľke. Kvalitná hash funkcia by mala byť rýchla na výpočet, pretože výkon hash funkcie ovplyvňuje výkon hash tabuľky. Ďalšia rovnako podstatná vlastnosť hash funkcie je jej diskrétne rovnomerné rozdelenie. Či už pri

otvorenom adresovaní alebo pri separátnom reťazení sa hash tabuľka značne spomaľuje ak sa elementy vkladajú na rovnaký index. Preto je diskrétne rovnomerné rozdelenie kľúčovou vlastnosťou všetkých hash funkcií. [13]

Ako perfektnú hash funkciu označujeme takú hash funkciu, ktorá rozloží všetky hodnoty do hash tabuľky bez spôsobenia žiadnych kolízií. Toto je možné dosiahnuť, ak je predom známy celý set hodnôt s ktorým bude hash tabuľka pracovať. [13]

$$h(x) = x \bmod m$$

Rovnica 3. Hash funkcia pomocou modula

Pri predpoklade, že kľúč je celočíselná hodnota, je možné použiť jednoduchú operáciu modulo na hodnotu kľúča na získanie indexu. Pomocou tejto operácie jednoducho získame index pre pár v hash tabuľke. Modulo operácia sa bežne používa po aplikovaní inej hash funkcie. Ak používame operáciu modulo na výpočet ideálneho indexu, tak ju nie je nutné aplikovať po druhý krát. Výhodou tohto prístupu je jednoduchosť a fakt, že modulo operácia sa často využíva aj po aplikovaní inej hash funkcie, čo pri tomto prístupe ušetrí jeden krok. Nevýhodou je, že ak kľúče pozostávajú z hodnôt *m*, *m**2, *m**3 a podobne, tak sa stráca vlastnosť diskrétneho rovnomerného rozdelenia a výkon tejto hash funkcie začína degradovať. [13]

Často využívaná technika pri hashovaní menších kľúčov je hashovacia funkcia identity. V praxi to znamená, že sa hodnota prevedie do celočíselného tvaru a toto číslo sa použije ako hash. V tomto prípade je taktiež nutné aplikovať modulo operáciu na výsledné číslo na vypočítanie konkrétneho indexu v hash tabuľke.

Pre kľúče dlhšie ako 8 bajtov sa používajú hash funkcie, ktoré hashujú celý reťazec bytov. Nejde však o kryptografický hash týchto bajtov. Tieto hash funkcie majú za úlohu vytvoriť čo najkvalitnejší hash na kvalitné rozloženie elementov v hash tabuľke. Medzi tieto hash funkcie patria napríklad wyhash32 a murmurHash32, ktoré budú použité v praktickej časti diplomovej práce. Tieto hash funkcie zvyčajne využívajú bitové posuny, xor a iné bitové operácie na vypočítanie hashu. Na vypočítaný hash je taktiež nutné aplikovať operáciu modulo, aby bolo možné získať index v hash tabuľke.

$$h(x) = x \,\&\, (m - 1)$$

Rovnica 4. Rýchle modulo

Operácia modulo je široko využívaná vo vyhľadávaní v hash tabuľkách. Táto operácia je však pomerne pomalá a tak vznikla snaha ju nahradiť rýchlejšími, bitovými operáciami. Modulo je možné nahradiť bitovou operáciou AND v prípade, že *m* je mocnina dvojky.

2.3 Zhoda najdlhšieho prefixu

Zhoda najdlhšieho prefixu, anglicky Longest Prefix Match (LPM), je algoritmus, ktorý sa používa v sieťových čipoch pri IP smerovaní. Algoritmus použije cieľovú adresu IP na prehľadanie Forwarding Information Base (FIB), aby našiel všetky zodpovedajúce záznamy. Pri určovaní, či je záznam zhodný alebo nie, LPM používa dĺžku prefixu uloženú vo FIB na zamaskovanie bitov cieľovej adresy, na ktorých pri porovnávaní nezáleží. [14]

#	IP adresa	Maska	Port
1	192.168.0.0	255.255.0.0	Eth0
2	192.168.1.0	255.255.255.0	Eth1
3	192.168.13.128	255.255.255.128	Eth2
4	192.168.13.0	255.255.255.0	Eth3

Tabuľka 1. Príklad FIB

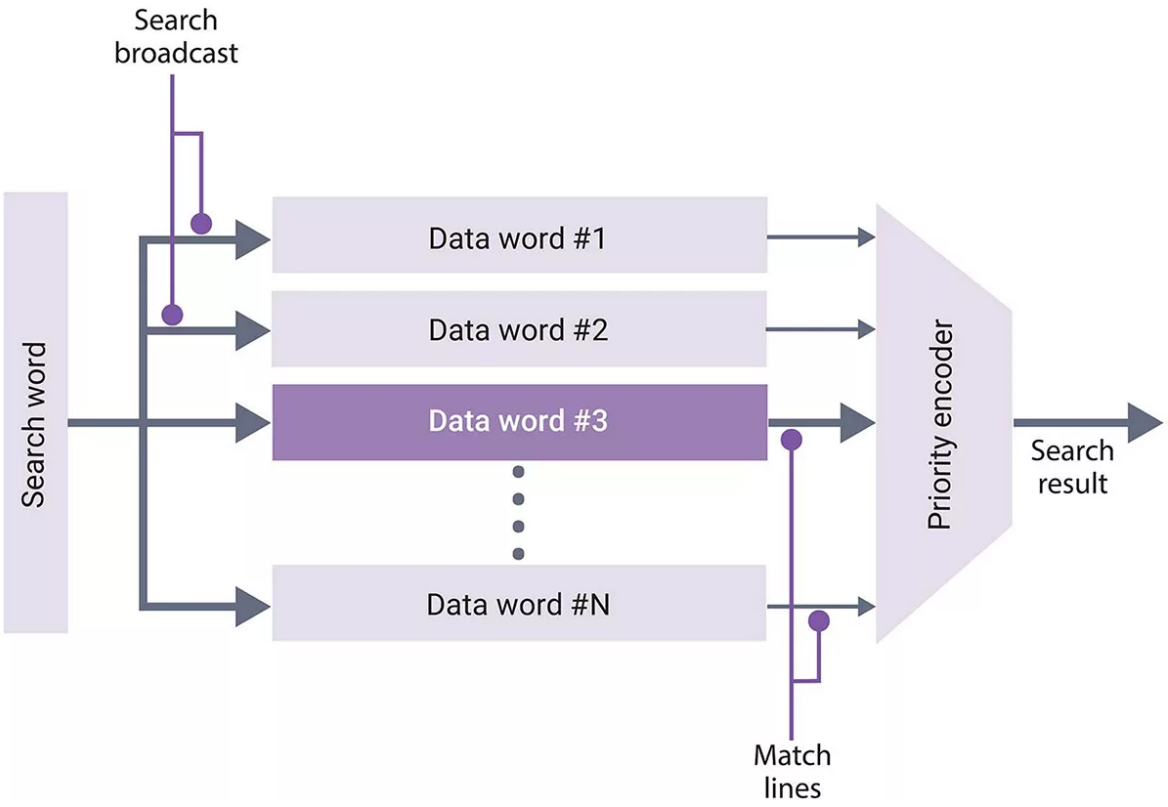
Ak je cieľová adresa napríklad 192.168.13.130, tak sa v tabuľke 1 zhoduje so záznamami 1, 3 a 4. Úlohou algoritmu LPM je vybratie adresy s najdlhším prefixom z týchto záznamov. Pri zobrazení IP adries v binárnej reprezentácií môžeme vidieť, že najdlhší prefix má IP adresa v zázname 3.

Hľadaná IP adresa - 11000000.10101000.00001101.10000010
IP adresa 1. 11000000.10101000.00000000.00000000
IP adresa 3. 11000000.10101000.00001101.10000000
IP adresa 4. 11000000.10101000.00001101.00000000

V zjednodušenom príklade, ktorý bol uvedený vyššie, boli vo FIB tabuľke uvedené iba 4 záznamy. Databázy FIB v typických routeroch však obsahujú milióny záznamov a požiadavky na uloženie väčšieho počtu záznamov v hardvéri naďalej rastú. Okrem toho, výrobcovia sieťových čipov do každého čipu vkladajú čoraz väčšiu šírku pásma, čo predstavuje veľkú záťaž pre zariadenia na spracovanie paketov, ktoré musia tieto vyhľadávania LPM vykonávať veľmi vysokou rýchlosťou.

2.4 TCAM

Ternary Content Addresable Memory (TCAM) je druh pamäte, ktorý je často využívaný ako súčasť Ethernetového prepínača alebo routera. Vyhľadávanie pomocou TCAM je teda zakorenené v hardvéri zariadenia. Predtým, ako bude možné vysvetliť čo to TCAM je, bude nutné popísať Content Addresable Memory (CAM) všeobecne. [15]



Obrázok 4. Vyhľadávanie v CAM [16]

CAM je asociatívna pamäť, ktorá sa od pamäte Random Access Memory (RAM) odlišuje tým, že k adresovaniu pamäte pristupuje iným spôsobom. Pri vyhľadávaní v tradičnej RAM sa vráti obsah pamäťových buniek z určenej adresy. Pri práci s CAM ale nie je poznané konkrétne miesto v pamäti. Známy je však obsah, ktorý sa snažíme vyhľadať. Ak by bol obsah nájdený v pamäti, tak CAM vráti list jednej alebo viacero adries kde bol tento obsah nájdený. Ide o náročnejšiu formu adresovania oproti pamäti RAM. CAM sa považuje za hardvérovú implementáciu hash tabuliek. Lineárne prehľadávanie zoznamu je podobné prechádzaniu všetkých miest pamäte a porovnávaniu s kľúčom pri sériovom prehľadávaní. Vyhľadávanie na báze CAM je ekvivalent paralelného porovnávania so všetkými obsahmi a následného vrátenia adresy úspešného porovnania. [15] [16]

Spolu s výkonnostnými výhodami prichádzajú aj nevýhody v podobe väčšej plochy a vyššieho rozptylu energie. Na rozdiel od statickej pamäte RAM (SRAM), ktorá má jednoduché pamäťové bunky, každý jednotlivý pamäťový bit v plne paralelnej pamäti CAM musí mať vlastný súvisiaci porovnávací obvod na zistenie zhody medzi uloženým bitom a vstupným bitom. Okrem toho sa pri použití CAM musia kombinovať výstupy zhody z každej bunky v dátovom slove, aby sa získal úplný signál zhody dátového slova. Dodatočné obvody zvyšujú fyzickú veľkosť CAM. V danom cykle je aktívne veľké množstvo obvodov, pretože všetky záznamy sa prehľadávajú paralelne. Preto je kľúčovou úlohou minimalizovať spotrebu energie CAM, ktorá rastie spolu s veľkosťou konfigurácie CAM. [16]

Existujú dva najznámejšie druhy CAM. BCAM je binárna pamäť CAM, ktorá podporuje binárne nuly a jednotky. TCAM predstavuje tretí (Ternary) stav *X* alebo taktiež nazývaný „DON'T CARE" stav, ktorý sa môže zhodovať s akoukoľvek hodnotou, nula alebo jedna. To znamená, že ternárna pamäť môže pracovať s troma rôznymi hodnotami: jedna, nula alebo *X*. [15] [16]

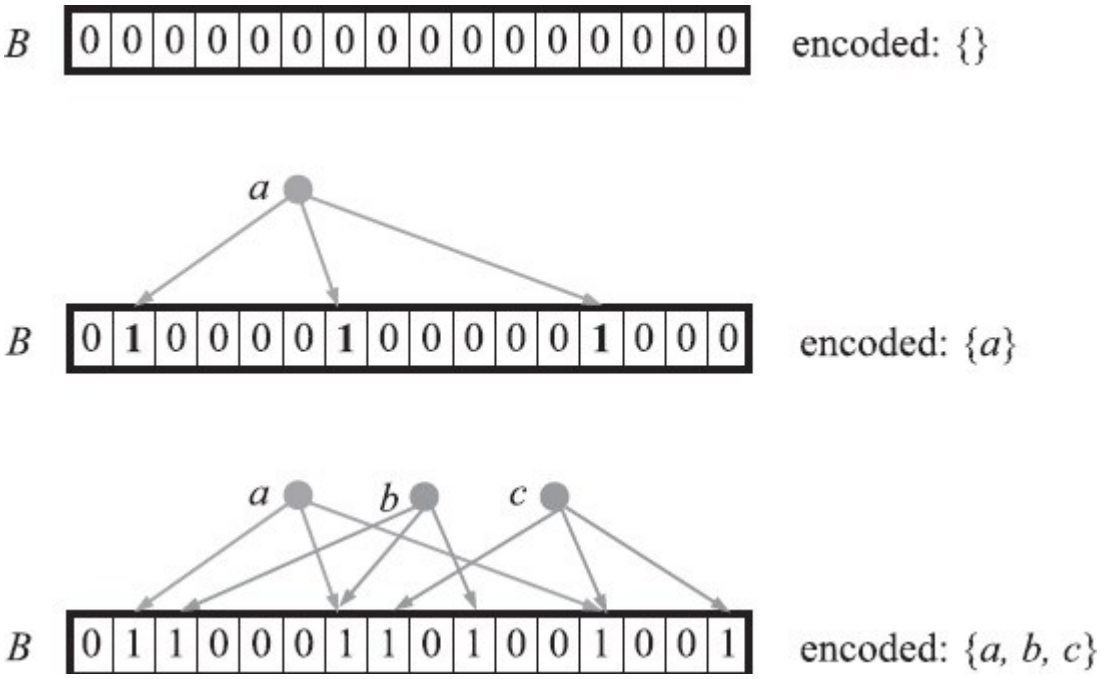
Hlavný problém s využitím TCAM je miesto, ktoré zaberá na silikóne. TCAM vyžaduje okolo 16 tranzistorov na pamäťovú bunku. Pre porovnanie, SRAM vyžaduje medzi 4 až 6 tranzistormi na pamäťovú bunku a DRAM iba jeden. Okrem toho, každá vyhľadávacia operácia zahŕňa aktiváciu všetkých záznamov, čím spotrebúvajú veľké množstvo energie na jedno vyhľadávanie. [14]

V ternárnej tabuľke môže byť jeden vyhľadávací kľúč zhodný s viacerými záznamami tabuľky, a preto musí mať každý záznam prioritu pridelenú softvérom. Ak sa zhoduje viacero záznamov, ako zodpovedajúci záznam sa vráti záznam s najvyššou prioritou. Ternárne porovnávacie tabuľky sa zvyčajne implementujú pomocou ternárnej pamäte adresovateľnej obsahom. TCAM poskytuje deterministické a vysokorýchlostné vyhľadávanie. Správu pamäte TCAM je možné implementovať dvoma spôsobmi, hardvérovým a softvérovým. Pri pridávaní položky s použitím hardvérovej správy pamäte, softvér dodáva kľúč a špecifickú prioritu pre túto položku. Hardvér potom nájde správne umiestnenie položky v tabuľke TCAM a v prípade potreby premiestni existujúce položky na základe priority novo pridanej položky. Pri pridávaní položky s použitím softvérovej správy pamäte, dodáva kľúč softvér a konkrétne miesto, kam sa má položka pridať. Hardvér potom pridá položku na presné miesto určené softvérovým príkazom.

2.5 Bloom filtre

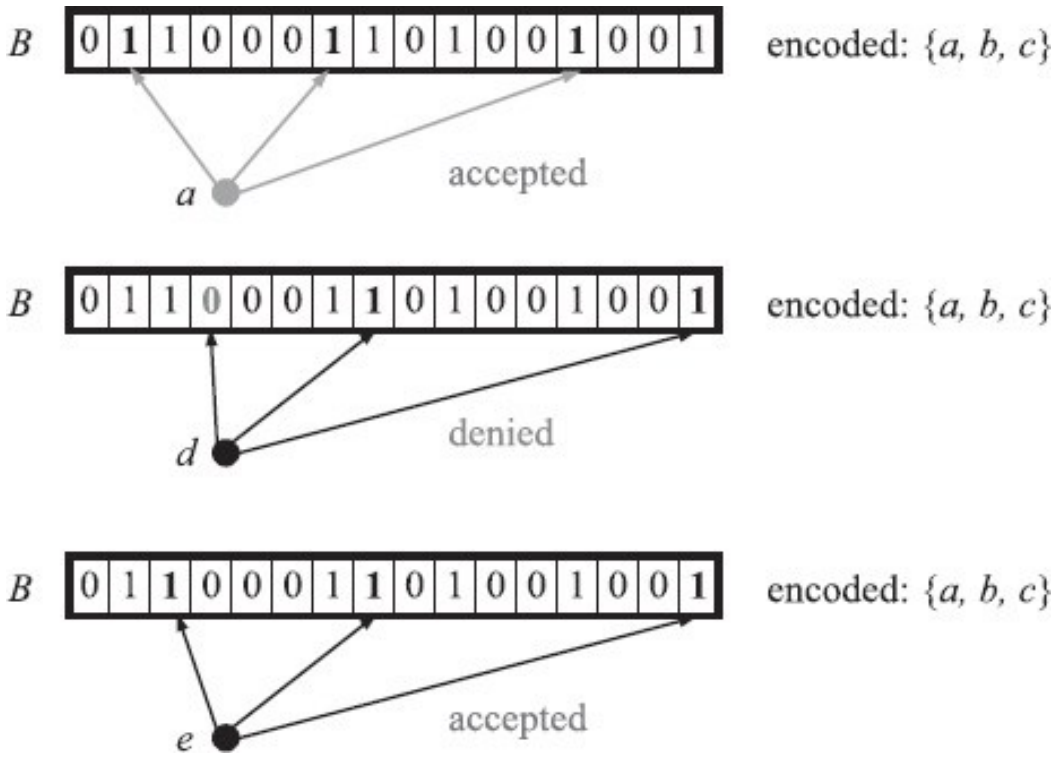
Bloom filter je kompaktná dátová štruktúra na rýchlu kontrolu príslušnosti k množine údajov. Prvýkrát ho navrhol Burton Howard Bloom v roku 1970. Vyhľadávanie prvku Bloom filter môže dať falošne pozitívne výsledky, ale falošne negatívne nie sú možné.

Bloom filter predstavuje pole bitov, ktoré je inicializované nulami. Podobne ako hash tabuľky, Bloom filter taktiež využíva hash funkcie k vkladaniu hodnoty do poľa. Počet hash funkcií je konštanta *k*. [13]



Obrázok 5. Vkladanie do Bloom filtra [13]

Na obrázku 5. môžeme vidieť proces vkladania elementov do Bloom filtra. Na tomto obrázku je vidieť, že tento konkrétny Bloom filter využíva 3 hash funkcie na získanie indexov. Pri vkladaní elementu do Bloom filtra sa vkladaná hodnota zahashuje a získa sa index za každú hash funkciu. Takto získavame *k* indexov. V každom z týchto indexov je nastavená hodnota na jedna. [13]



Obrázok 6. Kontrola príslušnosti v Bloom filtri [13]

Pri kontrole príslušnosti elementu v Bloom filtri sa opäť prepočítajú hash funkcie a získavajú sa indexy. Ak na týchto indexoch ležia samé jednotky, tak je možné tvrdiť, že element sa „možno“ nachádza v Bloom filtri. Naopak ak je hoci jedna hodnota nula, môžeme s istotou tvrdiť že sa element nenachádza v Bloom filtri. [13]

Keďže sa v Bloom filtri elementy často prekrývajú, z Bloom filtra nie je možné odstrániť žiaden element bez narušenia jeho integrity, alebo znovu vložením všetkých elementov Bloom filtru. [13]

Na obrázku 6. môžeme vidieť ako prebieha vyhľadávanie v Bloom filtri. Prvý príklad v tomto obrázku popisuje bežný prípad, kedy sa hľadá element ktorý sa nachádza v Bloom filtri. Toto hľadanie je samozrejme úspešné, keďže písmeno „a“ je element Bloom filtra. Písmeno „d“ je však zamietnuté, pretože jedna hodnota na indexe získaným hash funkciou je nulová. Prípad písmená „e“ je falošne pozitívny výsledok. Písmeno „e“ sa nenachádza v Bloom filtri, ale všetky indexy získané hash funkciami obsahujú hodnotu jedna. [13]

2.6 Existujúce porovnania hash tabuliek

Hľadanie práce, ktorá sa zaoberá porovnávaním algoritmov pre Ethernet vyhľadávanie na embedded systémoch, bolo neúspešné. Avšak, pri tomto hľadaní sa mi podarilo nájsť benchmark hash tabuliek, ktoré sa často využívajú v Ethernet vyhľadávaní.

Benchmark, ktorý najviac pomohol pri vypracovaní diplomovej práce, je z roku 2022. To znamená, že niektoré výsledky už môžu byť zastarané. Kód benchmarku, ako aj jeho výsledky, sú voľne dostupné. Benchmark bol napísaný v jazyku C++ a porovnával 29 hash máp. Bol spustený na osobnom počítači s operačným systémom Manjaro Linux a procesor bol použitý Intel i7-8700, uzamknutý na 3200 MHz. Tento PC benchmark pomohol s výberom vhodných hash máp na porovnanie v benchmarku vykonanom v praktickej časti diplomovej práce. Algoritmy, ktoré boli atraktívne z hľadiska benchmarkovania na embedded RISC-V akcelerátore, boli tie najvýkonnejšie, ktoré by bolo možné na spomínanom embedded zariadení spustiť. [17]

Ostatné porovnania algoritmov, ktoré som bol schopný nájsť, boli buď príliš zastarané, alebo boli potencionálne neobjektívne, ako napríklad článok od Malte Skarupke, ktorý tvrdí, že jeho hash tabuľka je najrýchlejšia a zdrojový kód, ktorý bol použitý pre tento benchmark, nie je voľne dostupný. [19]

2.7 Popis voľne dostupných algoritmov

Praktická časť tejto práce sa bude zaoberať porovnaním voľne dostupných algoritmov pre vyhľadávanie v Ethernete. V tejto časti je preto nutné popísať voľne dostupné algoritmy, ktoré budú ďalej zvážené na potencionálne využité v praktickej časti. Každý voľne dostupný algoritmus bude popísaný podľa techník, ktoré využíva, budú vyzdvihnuté unikátne vlastnosti každého algoritmu a zhrnuté výhody a nevýhody algoritmu. Algoritmy boli vybrané na základe existujúcich porovnaní hash tabuliek na osobných počítačoch a ich vhodnosti pre Ethernetový RISC-V akcelerátor.

2.7.1 static_flat_map

Static_flat_map bola vytvorená v rámci praktickej časti ako základ pre porovnanie s ostatnými voľne dostupnými algoritmami. Hash mapu by NXP Semiconductors mohlo používať bez obáv z prípadných legálnych problémov. Hash mapa je napísaná v jazyku C, používa otvorené adresovanie, Robin Hood hashovanie. Bola inšpirovaná mapou flat_hash_map od Malte Skarupke. Robin Hood hashovanie s využitím lineárneho

próbovania nám poskytuje dobrú pamäťovú lokalitu, pretože elementy sú umiestnené v poli a sú prechádzané lineárne. Táto tabuľka bola navrhnutá pre statickú alokáciu pamäte, kvôli implementácií v embedded systémoch. Tým pádom ju po inicializácií nie je možné rozširovať na viac elementov dynamicky. Základná hodnota maximálneho koeficientu zaťaženia je nastavená na 0.5.

Výhody:

- dobrá cache lokalita
- žiadne licenčné problémy
- možnosť statickej alokácie pamäte

Nevýhody:

- implementovaná jedným človekom - možnosť výskytu chýb
- pravdepodobne pomalšia ako ostatné, sofistikovanejšie algoritmy
- zaberá viac miesta v porovnaní s ktoroukoľvek mapou ktorá používa separátne reťazenie

2.7.2 unordered_map

Unordered_map predstavuje C++ štandard pre asociatívne kontajnery. Je definovaná v namespace std. Vnútorne prvky nie sú zoradené v žiadnom konkrétnom poradí, ale sú usporiadané do bucketov. Do ktorého bucketu sa prvok zaradí, závisí výlučne od hashu konkrétneho kľúča. Kľúče s rovnakým hashom sa zobrazujú v rovnakom buckete. Bucket v unordered_map je hlava lineárneho zoznamu. Táto mapa teda spadá medzi hash tabuľky so separátnym reťazením. Základná hodnota maximálneho koeficientu zaťaženia je nastavená na 1. Ako predvolenú hash funkciu používa std::hash. [18]

Výhody:

- C++ štandard, dobre udržiavaná
- žiadne licenčné problémy
- jednoduché použitie
- referenčná stabilita

Nevýhody:

- horšia cache lokalita, keďže ide o pole lineárnych listov

- pravdepodobne pomalšia ako ostatné, keďže musí zachovať referenčnú stabilitu

2.7.3 flat_hash_map

Táto hash mapa používa rovnaké techniky ako static_flat_map ktorá bola inšpirovaná touto mapou. Táto mapa je napísaná v jazyku C++ a je obsiahnutá v jednom hlavičkovom súbore. Používa otvorené adresovanie s lineárnym próbovaním. Rovnako používa Robin Hood hashovanie. Na dynamické rozširovanie veľkosti hash tabuľky sa predvolene využívajú prvočísla. Tabuľka taktiež podporuje rozširovanie veľkosti hash tabuľky pomocou mocniny dvojky. Táto tabuľka je výnimočná tým, že implementuje horný limit na počte próbovaných elementov. Ak je tento limit prekročený, vyhľadávanie zlyhá. Toto zrýchľuje vyhľadávanie v prípadoch neúspešného vyhľadávania. Základná hodnota maximálneho koeficientu zaťaženia je nastavená na 0.5. Ako predvolenú hash funkciu používa std::hash. [19]

Výhody:

- jednoduché použitie, stačí pridať hlavičkový súbor
- dobrá cache lokalita

Nevýhody:

- implementovaná jedným človekom – vyššia možnosť výskytu chýb
- neudržiavaná – bola implementovaná v roku 2017 a posledná úprava bola urobená v roku 2018
- zaberá viac miesta už aj podľa benchmarkov jej autora [19]

2.7.4 dense_hash_map

Voľne dostupná hash mapa vytvorená spoločnosťou Google. Je napísaná v jazyku C++, a je zahrnutá v Google projekte sparsehash, ktorý obsahuje viaceré implementácie hash máp. Dense_hash_map je podľa dokumentácie najrýchlejšia z máp tejto knižnice, avšak s najväčšou pamäťovou réžiou. Hash tabuľka dense_hash_map používa otvorené adresovanie a na prechádzanie mapy je použité kvadratické próbovanie. Základná hodnota maximálneho koeficientu zaťaženia je nastavená na 0.5. Ako predvolenú hash funkciu používa std::hash. [20]

Výhody:

- teoreticky veľmi rýchla hash tabuľka

- implementovaná korporáciou, menšia náchylnosť na chyby v implementácií

Nevýhody:

- relatívne stará implementácia, nebola upravená približne 8 rokov
- vysoká pamäťová réžia
- zložitejšie použitie, nutnosť zahrnutia viacerých súborov, nie len jedného hlavičkového súboru

2.7.5 F14

F14 je próbovacia hashovacia tabuľka od firmy Meta, pôvodne známej ako Facebook. F14 je napísaná v jazyku C++ a je súčasťou knižnice od firmy Meta s názvom folly, čo je voľnejší akronym pre Facebook Open Source Library. F14 rieši kolízie dvojitým hashovaním. V jednom „chunku“ (t.j. buckete) na jednej pozícii hashovacej tabuľky je uložených až 14 kľúčov. Na filtrovanie v rámci chunku sa používajú vektorové inštrukcie. Vyhľadávanie v rámci chunku zaberie len niekoľko inštrukcií. F14 odkazuje na skutočnosť, že algoritmus filtruje až 14 kľúčov súčasne. Táto stratégia umožňuje prevádzkovať hashovaciu tabuľku s vysokým maximálnym koeficientom zaťaženia, pričom sú próbovacie reťazce veľmi krátke. [21] [22]

Výhody:

- implementovaná korporáciou, menšia náchylnosť na chyby v implementácií
- unikátny prístup s chunkami o veľkosti 14
- stále pravidelne upravovaná a udržiavaná

Nevýhody:

- zložitejšie použitie, nutnosť zahrnutia celej folly knihovne
- nie je vhodná pre všetky embedded systémy, ARM procesory sú podporované

2.7.6 robin_map

Táto mapa je súčasťou knižnice robin-map implementovanou autorom Tessil. Knižnica je voľne dostupná a implementovaná v jazyku C++ s buildovacím systémom CMake. Tessil implementuje viacero dátových štruktúr, táto práca však porovnáva tsl::robin_map. Táto hash mapa využíva otvorené adresovanie s lineárnym próbovaním a Robin Hood hashovaním. Z toho vyplýva, že teoreticky bude veľmi podobná flat_hash_map od Malte Skarupke a static_flat_map. Na dynamické rozširovanie veľkosti

hash tabuľky sa využívajú mocniny dvojky. Tabuľka taktiež podporuje rozširovanie veľkosti hash tabuľky pomocou prvočísel. Základná hodnota maximálneho koeficientu zaťaženia je nastavená na 0.5. Ako predvolenú hash funkciu používa std::hash. [23]

Výhody:

- stále pravidelne aktualizovaná
- dobrá cache lokalita
- jednoduché použitie, stačí pridať hlavičkový súbor, možnosť buildu s CMake

Nevýhody:

- vysoká pamäťová réžia

2.7.7 hopscotch_map

Podobne ako robin_map, hopscotch_map je implementovaná autorom Tessil a je súčasťou knižnice hopscotch-map. Knižnica je voľne dostupná a implementovaná v jazyku C++ s buildovacím systémom CMake. Táto hash mapa využíva otvorené adresovanie s lineárnym próbovaním a Hopscotch hashovaním. Na dynamické rozširovanie veľkosti hash tabuľky sa využívajú mocniny dvojky. Tabuľka taktiež podporuje rozširovanie veľkosti hash tabuľky pomocou prvočísel. Základná hodnota maximálneho koeficientu zaťaženia je nastavená na 0.5. Ako predvolenú hash funkciu používa std::hash. [24]

Výhody:

- teoreticky veľmi rýchla hash tabuľka
- stále pravidelne aktualizovaná
- dobrá cache lokalita
- jednoduché použitie, stačí pridať hlavičkový súbor, možnosť buildu s CMake

Nevýhody:

- vysoká pamäťová réžia

2.7.8 emhash8

Emhash8 je súčasťou voľne dostupnej emhash knižnice, ktorá obsahuje viacero emhash tabuliek s rôznym zameraním. Emhash8 je knižnica, ktorá využíva iba hlavičkový súbor pre svoju kompletnú implementáciu. Nepoužíva tradičnú metódu na usporiadanie svojich elementov. Dáta sú uložené v jednom poli s tým, že každý element poľa je štruktúra.

Kolízia je riešená tak, že algoritmus sa chová ako keby používal separátne reťazenie, ale všetko je uložené v spomínanom poli. Podporuje lineárne a kvadratické próbovanie. Základná hodnota maximálneho koeficientu zaťaženia je nastavená na 0.8. Ako predvolenú hash funkciu používa std::hash, pre kľúče typu std::string používa voľne dostupný hash algoritmus tretej strany wyhash. [25]

Výhody:

- stále pravidelne aktualizovaná
- dobrá cache lokalita pri lineárnom próbovaní
- jednoduché použitie, stačí pridať hlavičkový súbor

Nevýhody:

- niektoré verzie emhash používajú x86 inštrukčný bit scanf(ctz), čo nemusí byť dostupné na všetkých architektúrach [25]

2.7.9 libcuckoo

Knižnica libcuckoo implementuje voľne dostupnú hash tabuľku, ktorá využíva cuckoo hashovanie. Táto hash mapa je konkurentná (súbežná), dokáže pracovať s viacerými zapisujúcimi a čítacími vláknami. Je implementovaná v C++ ale poskytuje rozhranie pre použitie v jazyku C. Hash tabuľka používa otvorené adresovanie. Rozhraním sa odlišuje od ostatných máp zahrnutých v tejto práci tým, že nenapodobňuje rozhranie std::unordered_map. Základná hodnota maximálneho koeficientu zaťaženia nie je definovaná. Ako predvolenú hash funkciu používa std::hash. [26]

Výhody:

- poskytnuté rozhranie pre programovací jazyk C
- jednoduché použitie, stačí pridať hlavičkový súbor (neplatí pri použití v C)

Nevýhody:

- niektoré embedded systémy nemusia podporovať viac jadrový charakter tejto tabuľky
- nebola aktualizovaná od 2022

3 POPIS TECHNOLÓGIÍ RISC-V AKCELERÁTORU OD NXP

V tejto kapitole budú vysvetlené potrebné komponenty, ktoré sú významnou súčasťou Ethernet akcelerátora od NXP, ktorý slúži k zrýchleniu spracovania, odosielania a preposielania paketov v Ethernete a zároveň odľahčuje hostiteľský procesor. V tejto časti práce budú popísané offloady a akcelerátory, ktoré pomáhajú odľahčiť prácu z hostiteľského procesoru. Rozdiel medzi offloadom a akcelerátorom je v tom, že offload deleguje úlohy z CPU na špecializovaný hardvér s cieľom znížiť CPU réžiu, takzvane ho „odľahčuje“. Akcelerácia nie len odľahčuje procesor, ale taktiež zahŕňa špecializované hardvérové komponenty, ktoré špecificky zvyšujú výkon určitých sieťových funkcií. Taktiež bude krátko popísaná architektúra inštrukčnej sady RISC-V. Na záver budú vysvetlené pamäte cache a MESI protokol. Cache pamäte sú prítomné v každom modernom systéme, ale pomer cache miss/hit významne ovplyvňuje výkon algoritmov pre vyhľadávanie v Ethernete.

3.1 Offloady

Offload sa vzťahuje na proces delegovania určitých úloh z Central Processing Unit (CPU) na špecializovaný hardvér alebo firmvér s cieľom zvýšiť výkon alebo efektivitu. V Ethernete sa offloading bežne týka úloh akými sú napríklad: výpočet kontrolného súčtu TCP/IP, odľahčenie TCP segmentácie, výpočet kontrolného súčtu STCP, hashovanie prichádzajúcich packetov, Transport Layer Security offloadovanie pre prichádzajúce a odchádzajúce packety a offloading Media Access Control Security operácií. Tieto techniky offloadingu pomáhajú odľahčiť CPU presunutím niektorých úloh na vyhradené hardvérové komponenty, ako sú karty sieťového rozhrania. [27]

Ako príklad offloadu bude využité spomínané odťaženie segmentácie rámca TCP. Segmentácia TCP rámca umožňuje zariadeniu segmentovať jeden rámec na viacero menších rámcov. Keď je TCP Segmentation Offload povolený, segmentáciu nebude vykonávať Linux kernel, ale iba odovzdá túto úlohu sieťovej karte. Kernel zasiela veľký nesegmentovaný rámec do sieťovej karty spolu s inštrukciami na segmentáciu. Sieťová karta, ktorá je optimalizovaná na sieťovú komunikáciu, prevezme tieto inštrukcie a rozdelí ich na menšie segmenty podľa špecifikácií protokolu TCP. Týmto sa uvoľní hostiteľský procesor, ktorý sa môže efektívne venovať iným procesom, pretože je odľahčený od segmentácie rámca TCP.

3.2 Akcelerátory

Hardvérové akcelerátory sa špecializujú na konkrétne funkcie. Odľahčujú pracovné zaťaženie z procesora a zároveň tieto konkrétne funkcie vykonávajú rýchlejšie než hostiteľský procesor. Keďže procesory sú navrhnuté na zvládanie širokej škály pracovných úloh, architektúry procesorov sú len zriedka optimálne pre konkrétne funkcie alebo pracovné záťaže. Hardvérové akcelerátory prinášajú výhody ako zlepšenie spotreby energie, výkonu a plochy systémov založených na procesoroch nezávisle od škálovania polovodičových procesov. [28]

Hardvérové akcelerátory sú niekedy nazývané ako koprocesory. Na rozdiel od offloadov, hardvérové akcelerátory nepotrebujú často komunikovať s hostiteľským procesorom k vykonávaniu ich činnosti. Hardvérové akcelerátory sú špecializované na určitú úlohu, ktorú v systéme vykonávajú. Z toho vyplýva, že akcelerátory nie len odľahčujú hostiteľský procesor, ale taktiež vykonávajú svoju špecializovanú úlohu rýchlejšie ako procesor určený na všeobecné účely.

Hardvérové akcelerátory sa v praxi využívajú na mnoho účelov. Medzi najznámejšie hardvérové akcelerátory patria General Purpose Graphical Procesing Unit (GPGPU), kryptografické akcelerátory, sieťové akcelerátory, Network Interface Card, Digital Signal Processor, akcelerátory umelej inteligencie a ďalšie.

GPGPU sú typy akcelerátorov, ktoré sa využívajú na všeobecné využitie. Nie sú konkrétne špecializované, a preto je možné ich využiť na širokú škálu úloh. Avšak, nie sú optimalizované pre konkrétne úlohy. Najviac relevantné sú pre túto prácu sieťové akcelerátory.

Sieťové akcelerátory pomáhajú zrýchľovať systém tým, že rýchlo a efektívne spracúvajú sieťovú prevádzku a tým zároveň uľahčujú pracovné zaťaženie procesoru. Ako praktické príklady sieťových akcelerátorov je možné uviesť Ethernet prepínač SJA1110 pre TSN alebo S32G Packet Forwarding Engine (PFE) od firmy NXP. Hardvérový produkt SJA1110 od NXP je mierený na využitie v automobilovom priemysle. SJA1110 rieši smerovanie, firewall, prevenciu a detekciu proti nežiadanému vniknutiu, pokročilý manažment Ethernet rámcov pomocou pamäte TCAM a mnoho ďalších funkcií. [29]

PFE je hardvérový akcelerátor, ktorý je taktiež navrhnutý na využitie v automobilovom priemysle. PFE poskytuje funkcionalitu IPv4/IPv6 routera a L2 most. Most na úrovni L2 poskytuje funkcionalitu presmerovania Ethernet paketov na základe ich MAC adries. PFE L2 most poskytuje možnosť presunutia úloh súvisiacich s mostíkmi z

hostiteľského CPU na PFE, a tým odľahčuje hostiteľský sieťový zásobník. Router IPv4/IPv6 je špecializovaná funkcia PFE na odľahčenie hostiteľského procesora od úloh súvisiacich s presmerovaním špecifickej IP prevádzky medzi dvoma fyzickými rozhraniami. Za normálnych okolností sa vstupná IP prevádzka odovzdáva CPU s TCP/IP stackom, ktorý je zodpovedný za smerovanie štandard paketov. Keď TCP/IP stack identifikuje, že paket nepatrí žiadnemu z miestnych koncových bodov IP, vykoná vyhľadávanie v smerovacej tabuľke, aby určil ako spracovať takúto prevádzku. PFE môže byť nakonfigurovaný tak, aby identifikoval toky, ktoré nemusia vstúpiť do hostiteľského CPU. Toto sa vykoná pomocou jeho internej smerovacej tabuľky. Zabezpečí tak, aby správne pakety boli presmerované na správne cieľové rozhranie. [30]

3.3 RISC-V

RISC-V je architektúra inštrukčnej sady, ktorá bola pôvodne navrhnutá na podporu výskumu a vzdelávania v oblasti počítačovej architektúry. V súčasnosti sa stáva aj štandardnou voľbou a otvorenou architektúrou pre priemyselné implementácie. [31]

Členskými spoločnosťami RISC-V International, je RISC-V definovaný ako otvorený štandard. Zámerom je, aby členské spoločnosti mohli prostredníctvom spolupráce prispieť k novým možnostiam inovácie procesorov a zároveň podporiť nové stupne slobody návrhu. Architektúra RISC-V umožňuje konštruktérom vytvoriť svoj procesor spôsobom, ktorý je prispôsobený ich cieľovým koncovým aplikáciám, takže môžu optimalizovať spotrebu, výkon a plochu pre tieto aplikácie. Instruction set architecture (ISA) je set inštrukcií, ktoré procesor môže vykonávať. RISC-V tiež poskytuje flexibilitu pri výbere z dostupných funkcií, namiesto toho, aby sa musel používať celý súbor funkcií. [32]

ISA RISC-V je definovaná ako ISA s celočíselným základom, ktorá musí byť prítomná v každej implementácii. Ako dodatok môže byť rozšírená o voliteľné rozšírenia základnej ISA. Základná celočíselná ISA je veľmi podobná ISA prvých procesorov Reduced Instruction Set Computer (RISC) s výnimkou toho, že nemá sloty na oneskorenie vetiev a podporuje voliteľné kódovanie inštrukcií s premenlivou dĺžkou. Základ je starostlivo obmedzený na minimálnu sadu inštrukcií, ktorý postačuje na to, aby poskytol rozumný cieľ pre kompilátory, asemblery, linkery a operačné systémy, čím poskytuje vhodný základ ISA a softvérového reťazca nástrojov, okolo ktorej možno vytvoriť viac prispôsobených ISA procesorov. [31]

Procesor navrhnutý s architektúrou RISC-V, využíva zjednodušené inštrukcie na vykonávanie rôznych úloh. Umožňuje tiež konštruktérom vytvárať tisíce potenciálnych

vlastných procesorov, čo uľahčuje rýchlejšie uvedenie na trh. Rozšírenosť RISC-V architektúry šetrí čas potrebný na vývoj softvéru, keďže je architektúra dobre známa a existujú voľne dostupné knižnice a projekty ktoré pomáhajú rýchlemu vývoju. Medzi ďalšie výhody RISC-V patrí, jej otvorená povaha, ktorá umožňuje spoluprácu a inovácie v celom odvetví. Výhodou je taktiež spoločná ISA, ktorá pomáha uľahčiť vývoj softvéru, pretože všetky procesory by potenciálne mohli používať rovnakú architektúru. Konštruktéri môžu používať rovnaký základ ISA od jednoduchých embedded zariadení až po najväčšie superpočítače, čím prispôsobia svoje zariadenie potrebám trhu. Okrem toho, jej otvorený charakter znamená, že celú architektúru RISC-V je možné podrobne preskúmať vo verejnej sfére, čím sa eliminujú zadné dvierka a skryté kanály. [32]

3.4 Vyrovnávacia pamäť

Dôležitou súčasťou RISC-V akcelerátoru je taktiež vyrovnávacia pamäť, ktorá slúži ako oporná jednotka celkového výkonu akcelerátoru. Bude vysvetlené, čo to vyrovnávacia pamäť je, ako sa delí, používa, budú predstavené rôzne konfigurácie a nakoniec bude popísaný koherenčný protokol, ktorý je nutnou súčasťou každého viacjadrového systému.

Úlohou vyrovnávacej pamäte je poskytovať vyššiu rýchlosť oproti operačnej pamäti systému. Vyrovnávacia pamäť je situovaná v blízkosti procesora a obsahuje kópiu dát, ktoré sú umiestnené v operačnej pamäti. Pri jednom procesore sa často využíva viac ako jedna vyrovnávacia pamäť. Vyrovnávacie pamäte sa potom označujú ako L1, L2, až L3. Písmeno „L“ v tomto kontexte značí „level“, teda úroveň vyrovnávacej pamäte. Typicky platí, že vyššia úroveň (L1) vyrovnávacej pamäte je rýchlejšia a obsahuje menej pamäte v porovnaní s nižšou úrovňou (L2).

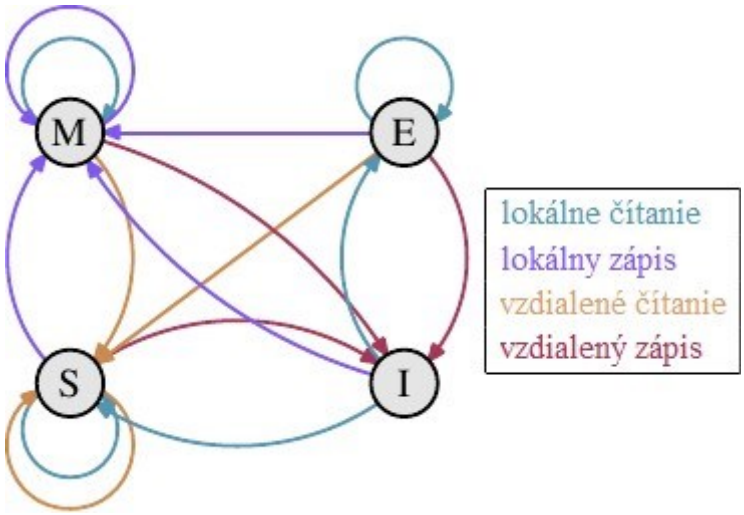
Vyrovnávacia pamäť môže byť unifikovaná alebo rozdelená pre osobitné ukladanie referencií na dáta a na inštrukcie. Typicky sa takto rozdeľuje vyrovnávacia pamäť na rovnakej úrovni, zvyčajne L1. Keď procesor, ktorý používa rozdelenú vyrovnávaciu pamäť potrebuje načítať inštrukcie z operačnej pamäte, obrátí sa najskôr na vyrovnávaciu pamäť úrovni L1-I. Ak rovnaký procesor potrebuje načítať dáta z operačnej pamäte, tak sa najskôr obráti na vyrovnávaciu pamäť úrovni L1-D. Hlavnou výhodou rozdelenej vyrovnávacej pamäte je, že eliminuje spor o vyrovnávaciu pamäť medzi jednotkou načítania/dekódovania inštrukcií a vykonávacou jednotkou. To je dôležité pri každom návrhu, ktorý sa spolieha na pipelining inštrukcií. Zvyčajne procesor načíta inštrukcie vopred a naplní vyrovnávaciu pamäť alebo pipeline inštrukciami, ktoré sa majú vykonať. [11]

3.4.1 MESI Protokol

Viacjadrové systémy potrebujú na zaistenie platnosti dát používať koherenčný protokol. Jedným z často využívaných protokolov je práve MESI protokol. Názov protokolu MESI je akronym, ktorý reprezentuje stavy, v ktorých sa môže riadok vyrovnávacej pamäte nachádzať. [33]

Tento akronym môžeme rozložiť na štyri stavy obsiahnuté v akronyme: Modify, Exclusive, Shared a Invalid. Stav Modify značí, že riadok vyrovnávacej pamäte bol modifikovaný procesorom a zároveň sa tento riadok nenachádza v žiadnej inej vyrovnávacej pamäti. Pre stav Exclusive je známe, že riadok vyrovnávacej pamäte nie je upravený a zároveň riadok nie je načítaný do vyrovnávacej pamäte žiadneho iného procesora, podobne ako v stave Modify. V stave Shared nie je riadok vyrovnávacej pamäte upravený a môže existovať vo vyrovnávacej pamäti iného procesora. A nakoniec stav Invalid, ktorý označuje riadok vyrovnávacej pamäte, ktorý je prázdny alebo neplatný. [33]

Tento protokol sa v priebehu rokov vyvinul z jednoduchších verzií, ktoré boli menej komplikované a aj menej účinné. Pomocou týchto štyroch stavov je možné efektívne implementovať vyrovnávacie pamäte so spätným zápisom a zároveň podporovať súbežné používanie údajov, ktoré sú určené iba na čítanie na rôznych procesoroch.



Obrázok 7. MESI protokol [33]

Prechody medzi rôznymi stavmi v protokole MESI môžeme vidieť na obrázku 7. Farbami sú znázornené možné prechody medzi MESI stavmi vyrovnávacej pamäti v prípade lokálneho a vzdialeného čítania a zápisu. Niektoré prechody medzi MESI stavmi je potrebné popísať, pretože viacjadrový systém sa bez koherenčného protokolu nezaobíde. [33]

Pri spustení zariadenia sú všetky riadky vyrovnávacej pamäte prázdne, a teda sú v stave Invalid. Ak sa do vyrovnávacej pamäte načítajú údaje na účel zápisu, tak sa stav vyrovnávacej pamäte zmení na Modified. Ak sú údaje načítané iba na čítanie, nový stav bude závisieť od toho, či má riadok vyrovnávacej pamäte načítaný aj iný procesor. Ak má riadok vyrovnávacej pamäte načítaný iba jeden procesor, stav vyrovnávacej pamäte sa zmení na Exclusive. Ak sa pri načítaní zistí, že riadok vyrovnávacej pamäte je načítaný aj pre iný procesor, tak sa ich stavy menia na Shared. [33]

Ak sa na lokálnom procesore číta alebo zapisuje riadok vyrovnávacej pamäte, ktorý je v stave Modified, procesor môže použiť aktuálny obsah vyrovnávacej pamäte a jej stav sa nezmení. Ak chce druhý procesor čítať z riadku vyrovnávacej pamäte, prvý procesor musí poslať obsah svojej vyrovnávacej pamäte druhému procesoru a potom môže zmeniť stav na Shared. Údaje odoslané druhému procesoru prijíma a spracováva aj pamäťový radič, ktorý ich obsah ukladá do pamäte. Ak by sa tak nestalo, riadok vyrovnávacej pamäte by nemohol byť označený ako zdieľaný. Ak chce druhý procesor zapisovať do riadka vyrovnávacej pamäte, prvý procesor odošle obsah riadka vyrovnávacej pamäte a lokálne označí riadok vyrovnávacej pamäte ako Invalid. Táto operácia sa nazýva „Request For Ownership“. Vykonanie RFO operácie v poslednej úrovni vyrovnávacej pamäte je nákladné. Pri zápise do L1 musíme pripočítať aj čas potrebný na zápis nového obsahu riadka L1 do L2 alebo do hlavnej pamäte, čo ďalej zvyšuje náklady. [33]

Ak je riadok vyrovnávacej pamäte v stave Shared a lokálny procesor z neho číta, nie je potrebná žiadna zmena stavu a požiadavka na čítanie sa môže splniť pomocou vyrovnávacej pamäte. Ak je riadok vyrovnávacej pamäte lokálne zapísaný, môže sa použiť aj riadok vyrovnávacej pamäte, ale stav sa zmení na Modified. Táto operácia však vyžaduje, aby všetky ostatné kópie riadku vyrovnávacej pamäte v iných procesoroch boli označené ako Invalid. Operácia zápisu musí ostatným procesorom oznámiť prostredníctvom správy RFO. Ak si riadok vyrovnávacej pamäte vyžiada na čítanie druhý procesor, nemusí sa nič stať. Hlavná pamäť obsahuje aktuálne údaje a lokálny stav je už zdieľaný. V prípade, že druhý procesor chce zapisovať do riadka vyrovnávacej pamäte, riadok vyrovnávacej pamäte sa jednoducho označí ako Invalid. [33]

Stav Exclusive sa od stavu Shared líši iba jediným závažným rozdielom. Ak je vyrovnávacia pamäť lokálneho procesora v stave Exclusive, tak lokálna operácia zápisu nemusí byť oznámená na zbernici. V tomto prípade je pre procesor známe, že lokálne kópia vyrovnávacej pamäte je jediná. Táto operácia je rýchlejšia a tým je zrejmé, že procesor sa bude snažiť udržať čo najviac riadkov vyrovnávacej pamäte v stave Exclusive namiesto

stavu Shared. Ten je núdzovým riešením v prípade, že informácie nie sú v danom okamihu k dispozícii. Stav Exclusive možno tiež úplne vynechať bez toho, aby to spôsobilo funkčné problémy. Utrpí tým len výkon, pretože prechod zo stavu Exclusive do stavu Modified je oveľa rýchlejší ako prechod zo stavu Shared do stavu Modified. [33]

II. PRAKTICKÁ ČÁST

4 ÚVOD DO PRAKTICKEJ ČASTI

V praktickej časti bude implementovaný benchmarkovací program. Pomocou tohto programu budú porovnané rôzne voľne dostupné algoritmy. Výsledkom bude objektívne odporúčanie algoritmu na využitie v RISC-V akcelerátore firmy NXP Semiconductors. Kód benchmarkovacieho programu, ako aj Python skript, bude voľne dostupný a ľahko rozšíriteľný o ďalšiu funkcionalitu. Tento program má za cieľ pomôcť ako firme NXP Semiconductors, tak aj iným subjektom porovnať a vybrať efektívny algoritmus pre embedded Ethernet akcelerátory.

4.1 Použité technológie

Na implementáciu benchmarku voľne dostupných algoritmov, ktoré by mohli byť použité na vysoko výkonné vyhľadávanie pre Ethernet v NXP RISC-V Ethernet akcelerátore, je nevyhnutné vybrať správne technológie. Požiadavky na tieto technológie budú prevažne rýchlosť, použiteľnosť a spoľahlivosť.

Väčšina voľne dostupných algoritmov, ktoré budú subjektom spomínaného benchmarku, sú napísané v programovacom jazyku C++, ktorý poskytuje vysokú rýchlosť, potrebnú na implementáciu takýchto algoritmov. Programovací jazyk C++ oproti jazyku C taktiež podporuje vysokú úroveň abstrakcie, akou sú napríklad triedy, čo uľahčuje proces generickej implementácie. Na druhú stranu, programovací jazyk C poskytuje rýchlosť, jednoduchosť a efektívne využívanie zdrojov, ktoré sú pre embedded systémy veľmi dôležité a aj preto je široko využívaný embedded zariadeniami. Firmware pre RISC-V Ethernet akcelerátor je aj z týchto dôvodov programovaný práve v programovacom jazyku C. Na benchmarkovanie voľne dostupných algoritmov bude v tejto práci použitý programovací jazyk C, pretože pre implementáciu benchmarku nie je potrebná abstrakcia jazyka C++. Využitie programovacieho jazyka C nám taktiež zaistí čo najbližšie výsledky v porovnaní s reálnym scenárom, ktorý by mohol nastať v spomínanom NXP RISC-V Ethernet akcelerátore. Pre použitie C++ algoritmov v jazyku C bude nutné implementovať Application Programming Interface (API), ktorá bude sprístupňovať C++ algoritmy pre použitie v benchmarkovacom programe.

Ako build systém bude pri implementácií využívaný CMake, ktorý poskytuje jednoduchý, široko rozšírený, kvalitne zdokumentovaný spôsob na build aplikácií. Veľká časť voľne dostupných algoritmov a Firmware pre RISC-V Ethernet akcelerátor používa

taktiež CMake, čo uľahčuje integráciu benchmarku, ktorý bude vytvorený v rámci tejto práce.

Implementovaný benchmark bude spustený na RISC-V hardvérovom simulátore. Simulátor poskytuje presné dáta o časovom behu aplikácie v nanosekundách, CPU cykloch, miss rate rôznych úrovní cache pamäte a podobné metriky, ktoré budú využívané v benchmarku. Hardvérový RISC-V simulátor podporuje C++ verziu 14. Kvôli nutnosti využitia tohto simulátoru vzniká potreba spomínaného Python skriptu.

Benchmark napísaný v jazyku C bude kvôli obmedzeniam RISC-V simulátoru nutné spúšťať viac krát, pre každý algoritmus, pre rôzne veľkosti vstupných dát a pre rôzne benchmark funkcie, ktoré budú na vybranom RISC-V akcelerátore implementované. Preto bude potrebné implementovať sekundárny program, ktorý bude benchmark spúšťať a zaznamenávať výsledky rôznych benchmarkov a poprípade bude vykresľovať z nazhromaždených dát tabuľky a grafy. Pre implementáciu sekundárneho programu bol zvolený programovací jazyk Python, ktorý je síce pomalší než programovací jazyk C alebo C++, za to však poskytuje silnú zbierku knižníc na správu a vykresľovanie dát do grafov.

4.2 Požiadavky na implementáciu

Jazyk C bol vybraný pre implementáciu benchmarku hlavne kvôli jeho rýchlosti, jednoduchosti a kompatibilite s hardvérovým simulátorom RISC-V. Benchmark musí byť rýchly a presný a to nám kombinácia hardvérového simulátoru RISC-V a jazyka C zaisťuje. Implementácia benchmarku bude podporovať spustenie benchmarku na viacerých jadrách pre vhodné funkcie určené iba na čítanie, akou je napríklad vyhľadávanie.

V benchmarku budú implementované viaceré metriky na porovnanie vysoko výkonných algoritmov pre vyhľadávanie v Ethernete. Taktiež bude vytvorený API medzi jazykmi C/C++ v prípade, že voľne dostupné algoritmy, ktoré boli vybrané na porovnanie, nebudú napísané v jazyku C. V prípade ak algoritmus napísaný v jazyku C++ a verzia jazyka bude vyššia ako 14, tak bude vybraná alternatíva tohto algoritmu napísaná v kompatibilnej C++ verzii. Keďže tento benchmark bude spustený na RISC-V akcelerátore, nebude možné používať špeciálne Single Instruction, Multiple Data (SIMD) inštrukcie, ktoré môžu byť dostupné v ARM a x86/64 architektúrach. Algoritmy, ktoré využívajú SIMD inštrukcie, budú z tohto benchmarku vynechané, avšak pri použití hardvéru, ktorý tieto inštrukcie podporuje, môžu poskytnúť značné zrýchlenie daného algoritmu.

Python skript, ktorý už bol spomenutý kvôli použitému jazyku v predchádzajúcej kapitole, bude nutné implementovať z dôvodu potreby mnohonásobného spustenia benchmarku. Tento Python skript bude externe spúšťať benchmarkovací algoritmus napísaný v jazyku C, s rôznymi nastaveniami konfigurácie, ktorá bude obsahovať nastavenia pre rôzne metriky, hash funkcie a vysoko výkonné algoritmy pre viacero počtov elementov vložených do daného algoritmu. Grafy budú vykreslené pomocou knižnice matplotlib.pyplot, na ktorých bude zrejmé, aké algoritmy sú najvýkonnejšie pre konkrétny benchmark. Pyhton skript bude ukladať výsledky do Excel súborov, kde v každom súbore bude vytvorené hodnotenie algoritmov podľa metrík, akými sú čas alebo cache miss. Nakoniec budú vytvorené dva Excel súbory, ktoré budú obsahovať všetky vyhodnotenia z ostatných Excel súborov a zároveň aj finálne hodnotenia na základe aritmetického priemeru.

V rámci tejto práce bude implementovaná hash tabuľka v jazyku C. Implementácia hash tabuľky síce nie je nevyhnutná pre porovnanie ostatných algoritmov, ale poskytuje dobrý základ pre benchmark popri std::unordered_map v C++. Táto hash mapa bude implementovaná pomocou známych voľne dostupných techník.

Ďalej bude taktiež implementovaný Bloom filter, ktorý potenciálne zrýchli neúspešné vyhľadávanie za nízku réžiu pamäte. Táto dátová štruktúra sa bude využívať sekundárne k hash tabuľke pre zrýchlenie spomínaného neúspešného vyhľadávania. Implementácia Bloom filtra v rámci benchmarku umožňuje odporučiť alebo zavrhnúť použitie Bloom filtra ako sekundárnu dátovú štruktúru na vybranom RISC-V akcelerátore.

5 ANALÝZA

V rámci časových obmedzení reálneho sveta nie je možné integrovať každý voľne dostupný algoritmus do benchmarku na hardvérovom RISC-V simulátore. Preto je dôležité správne analyzovať a vybrať potenciálne najrýchlejšie voľne dostupné algoritmy, ktoré sú použiteľné na spomínanom hardvéri. Teoretická časť tejto práce poukázala na to, že pri vyhľadávaní v Ethernete je najhlavnejšia metrika rýchlosť algoritmu. Keďže sa táto práca týka akcelerátoru, ktorý bude integrovaný v embedded systéme, musíme brať taktiež ohľad na režijné náklady na pamäť, ktorý bude tento algoritmus spotrebovávať.

Pamäť TCAM umožňuje rýchle vyhľadávanie v Ethernete hardvérovým spôsobom. Použitím TCAM pamäte na silikóne je možné zaistiť veľmi rýchle vyhľadávanie v Ethernete. TCAM má však aj svoje nevýhody, medzi ktoré patrí väčšia plocha zabratého silikónu a vyšší rozptyl energie. Veľká časť embedded systémov si nemôže dovoliť využitie pamäte TCAM a musia sa spoliehať na softvérové riešenia. Toto je taktiež prípad RISC-V Ethernet akcelerátoru od firmy NXP Semiconductors, ktorému sa venuje praktická časť. Aké softvérové alternatívy je možno použiť namiesto pamäte TCAM?

Hash tabuľky môžeme považovať za najjednoduchšie softvérové riešenie. Existuje mnoho pozitív hash tabuliek pre využitie v Ethernet vyhľadávaní. Najpodstatnejšou výhodou je ich rýchlosť vyhľadávania O(1) v najlepšom prípade, čo už, ako bolo spomenuté, je veľmi podstatné pri Ethernet vyhľadávaní. Pozitívnym faktorom hash tabuliek je aj fakt, že sú dobre študované, existuje široký výber voľne dostupných hash tabuliek a sú jednoducho integrovateľné. Hash tabuľky taktiež poskytujú dobrú pamäťovú lokalitu, čo znamená rýchlejšie prístupy do pamäte a rýchlejší algoritmus pre Ethernet vyhľadávanie.

Bloom filter je dátová štruktúra, ktorá by mohla byť využitá na urýchlenie vyhľadávania v Ethernete. Bloom filtre neudržujú samotné dáta, držia iba informáciu o tom, či záznam v tabuľke možno existuje alebo neexistuje. V Bloom filtri nie je možné dostať „false negative“ stav, takže si môžeme byť istý že element v danom Bloom filtri neexistuje. Táto vlastnosť môže byť použitá pre urýchlenie vyhľadávanie v Ethernete, keď použijeme Bloom filter ako sekundárnu dátovú štruktúru. Pri takomto použití by sme vedeli zachytiť vyhľadávania, ktoré nie sú uložené v primárnej dátovej štruktúre už v Bloom filtri a tým by sa zrýchlili neúspešné vyhľadávania za cenu režijných pamäťových nákladov pre Bloom filter.

5.1 Analýza navrhnutých algoritmov

V teoretickej časti tejto práce boli popísané voľne dostupné algoritmy, ktoré sú vhodné pre vysoko rýchlostné vyhľadávanie v Ethernete. Táto podkapitola sa bude zaoberať analýzou týchto algoritmov. Bude zvážená ich vhodnosť pre porovnanie na RISC-V akcelerátore od NXP Semiconductors s ohľadom na jednoduchosť integrácie a limity hardvérového simulátoru. Hardvérový simulátor nepodporuje výnimky. Algoritmy, ktoré výnimky používajú boli upravené tak, aby namiesto vyhodenia výnimky ukončili svoju činnosť.

Static_flat_map bola napísaná ako pevný základ v jazyku C na porovnanie algoritmov praktickej časti tejto práce. Pre potrebu zaťaženia hash tabuľky nie je možné používať iba obmedzený, ale aj rýchly zásobník. Pre potrebu porovnania bolo potrebné pôvodne staticky alokovanú mapu prepísať na dynamickú alokáciu, čím sa priblížime ostatným porovnávaním algoritmom, ktoré sú napísané v C++ a používajú dynamickú alokáciu pamäte. Meno hash mapy bolo zachované aj napriek tomu, že sa statická alokácia pre účely porovnania nepoužíva.

Unordered_map, ktorá je štandardom pre asociatívne polia v jazyku C++, nám poskytuje alternatívny základ popri static_flat_map napísanej v jazyku C. Mapa je súčasťou C++ štandardu, a tým pádom je prispôsobená tak, aby fungovala na veľkom množstve architektúr. Kvôli tejto kvalitnej kompatibilite nie je potrebné hash mapu nijako upravovať, pretože nemá žiadne problémy s RISC-V architektúrou. Táto hash mapa je dobrým základom aj kvôli tomu, že ostatné voľne dostupné hash algoritmy kopírujú API tejto mapy. [18]

Flat_hash_map je rýchla hash mapa, ktorá je dobre zdokumentovaná a benchmarkovaná jej autorom. Autor vo svojom blogu tvrdí, že napísal najrýchlejšiu hash tabuľku. Toto tvrdenie, ako aj blog, boli napísané v roku 2016, čo môže znamenať, že jeho tvrdenie už nie je aktuálne. Avšak, táto hash tabuľka by stále mala poskytovať kvalitnú rýchlosť pre porovnanie s ostatnými hash mapami. Hlavnou nevýhodou hash tabuľky je pamäťová réžia, ktorá je vyššia v porovnaní s ostatnými hash tabuľkami. Hash mapa je vhodná na porovnanie, pretože je napísaná vo verzii C++, ktorá hardvérový simulátor RISC-V akcelerátor podporuje. Pri benchmarku je však nutné vypnúť používanie výnimiek v kóde hash tabuľky. [19]

Dense_hash_map je vysoko výkonná, voľne dostupná hash tabuľka od spoločnosti Google. Autor flat_hash_map ju vo svojom blogu považoval za najväčšieho konkurenta svojej tabuľky. Táto hash mapa je predchodca absl::flat_hash_map od Google. Táto novšia verzia však používa niektoré funkcie C++ 17, ale nie je podporovaná hardvérovým simulátorom RISC-V akcelerátoru. Dense_hash_map je kompatibilná verzia tejto vysokorýchlostnej hash mapy a používa výnimky, ktoré boli pre účely porovnania odstránené. [20]

F14 je voľne dostupná hash mapa od spoločnosti Meta, pôvodne Facebook. Táto hash mapa je kvalitne zdokumentovaná a benchmarkovaná. Spoločnosť Meta ju využíva vo viacerých oblastiach svojej spoločnosti a je súčasťou ich voľne dostupnej knižnice s názvom folly. Výkon hash mapy vyzerá sľubne, no napriek tomu ju v rámci tohto benchmarku nebude možné použiť, pretože benchmark bude spustený na hardvérovom simulátore RISC-V akcelerátore, ktorý neposkytuje inštrukcie využívané týmto algoritmom. Spomínané inštrukcie sú SSE2 na x86_64 architektúre a NEON na aarch64. [21]

Obe mapy robin_map a hopscotch_map od Tessil sú vysoko výkonné hash mapy s rôznymi prístupmi k riešeniu konfliktov v hash mape. Tieto mapy sú kvalitnou alternatívou na porovnanie s flat_hash_map a dense_hash_map. Robin_map aj hopscotch_map používajú CMake, čo uľahčuje integráciu s hardvérovým simulátorom. Sú kompatibilné s verziou C++, ktorú podporuje hardvérový simulátor. Pri integrácií týchto algoritmov bolo taktiež potrebné vypnúť výnimky, ktoré tieto algoritmy podporujú. [23] [24]

Emhash8 je voľne dostupná hash mapa, ktorá pochádza zo súboru emhash máp. Je obsiahnutá v jednom hlavičkovom súbore, čo uľahčuje jej integráciu, nepoužíva žiadne výnimky, a taktiež je kompatibilná s verziou C++, ktorú používa hardvérový simulátor. Emhash8 je teda možné použiť v benchmarku jednoducho a bez žiadnych ďalších úprav. [25]

Bloom filter je možné využiť ako sekundárnu dátovú štruktúru. Cieľom použitia Bloom filtra je zredukovať čas potrebný na vyhľadanie elementu, ktorý sa v primárnej dátovej štruktúre nevyskytuje. Zaberá málo miesta v pamäti, ale na rozdiel od hash tabuliek, nedokáže uchovať hodnoty. Bloom filter drží iba informáciu o tom, či sa kľúč elementu vo filtri môže nachádzať alebo sa v ňom určite nenachádza. Ak sa element určite nenachádza v Bloom filtri, vieme urýchliť neúspešné vyhľadávanie a okamžite ho ukončiť.

6 IMPLEMENTÁCIA

Cieľom praktickej časti tejto práce je implementovať benchmarkovací program pre vybraný RISC-V akcelerátor od firmy NXP Semiconductors. V tejto časti budú uvedené detaily o implementácií samotného benchmarku, ako aj API medzi programovacími jazykmi C++ a C. Táto API je potrebná, pretože voľne dostupné algoritmy využívajú hlavne programovací C++ pre svoju implementáciu. Ďalej bude popísaná implementácia základnej hash tabuľky static_hash_map, ktorá je potrebná ako licenčný a benchmarkovací základ tejto práce. Nakoniec bude priblížená implementácia Python skriptu, ktorého účelom je externe spúšťať benchmarkovací program s rôznou konfiguráciou, extrahovať a vykresľovať dáta z hardvérového simulátoru. Okrem detailov implementácie budú taktiež popísané problémy, ktoré počas implementácie vznikli a ich riešenia.

6.1 Implementácia benchmarku

Porovnanie vysoko výkonných algoritmov pre Ethernet na vybranom RISC-V akcelerátore bolo napísané v programovacom jazyku C, ktorý nám poskytuje jeho rýchlosť, jednoduchosť a predovšetkým reálne prostredie, v ktorom bude najlepší algoritmus tohto benchmarku používaný. Hardvérový simulátor zbiera štatistiky počas behu programu a umožňuje nám z nich čerpať dáta. V zdrojovom kóde sa okrem štandardných knižníc používa hlavičkový runtime súbor, ktorý poskytuje API hardvérového simulátoru. Toto API nám umožňuje využiť viacero jadier RISC-V akcelerátoru pre použitie v benchmarku.

Benchmarkovací program príma a spracúva vstupy z príkazového riadku. Tieto vstupy sú použité ako konfigurácia pre konkrétny benchmark. Ak je to možné, tak sa práca rozdelí medzi všetky dostupné jadrá hardvérového simulátoru. Prvé jadro vždy vykoná všetku potrebnú prípravu pre spustenie benchmarku. Do tejto prípravy je zahrnutá pseudonáhodná generácia kľúčov a hodnôt, inicializácia algoritmu a spomínané nastavenie konfigurácie na základe vstupov. Po nastavení konfigurácie a prípadnom rozdelení práce sa spustí žiadaný benchmark, ktorý je spustený funkciou, ktorá volá API vybraného algoritmu. Pomocou spomínaného runtime API sa pri zavolaní tejto funkcie taktiež spustí meranie v hardvérovom simulátore, z ktorého čerpáme dáta po vykonaní benchmarku. Po skončení benchmarku sa vypne meranie v hardvérovom simulátore a prvé jadro uvoľní všetku pamäť alokovanú pre vygenerované kľúče a hodnoty. V prípade, že nastane v benchmarkovacom procese chyba, prvé jadro je poverené uvoľnením alokovaných zdrojov.

Vstupy do benchmarkovacieho programu sa používajú ako konfigurácia samotného programu. Program vyžaduje tieto vstupy:

- množstvo dvojíc kľúč/hodnota
- index metriky na základe ktorej bude algoritmus porovnaný
- veľkosť kľúča v bajtoch
- veľkosť hodnoty v bajtoch
- index algoritmu ktorý bude použitý
- index hash funkcie ktorá bude použitá
- koeficient zaťaženia uvedený ako celočíselná hodnota
- použitý počet jadier

Spustením benchmarkovacieho programu s týmito vstupmi vieme získať presné hodnoty z hardvérového simulátoru RISC-V akcelerátoru na vykreslenie metriky pre konkrétny algoritmus a konkrétnu hash funkciu.

Konfigurácia benchmarku je uložená v štruktúre Benchmark, ktorá je uložená v globálnej pamäti programu. Štruktúra drží konfiguráciu pre spustený benchmark ako aj vygenerované hodnoty. Taktiež v sebe uchováva ukazovateľ na inštanciu algoritmu, s ktorým benchmark práve pracuje. Využitie globálnej pamäte sa často považuje za zlú praktiku, avšak jej použitie je nutné kvôli tomu, aby mali ostatné jadrá prístup k rovnakej konfigurácii a inštancii algoritmu. Program využíva na manažovanie jadier spomínanú runtime API, takže konvenčné používanie vlákien neplatí. Pseudonáhodné hodnoty pre kľúče a hodnoty ukladané algoritmom sa generujú vo funkcii generate_pairs, ktorá dynamicky alokuje pamäť a naplní polia „keys“ a „values“ s pseudonáhodnými reťazcami bajtov o dĺžke key_size a value_size. Funkcia switch_perf_modeling nám poskytuje vypnúť a zapnúť zber informácií pre jadro, na ktorom sa táto funkcia zavolá. Na začiatku programu sa táto funkcia zavolá pre okamžité vypnutie zberu informácií. Zber informácií znova zapneme iba pri spustení benchmarkovacej funkcie. Po jej skončení je znova vypnutá, aby sme získali čo najpresnejšie dáta. Pri inicializácií benchmarku prvé jadro importuje zvolený algoritmus pomocou kľúčového slova export. Implementácia metrík je definovaná v inom súbore. Okrem implementácie metrík a hlavného programu sa do projektu vkladajú viaceré hlavičkové súbory, ktoré poskytujú abstrakciu pre benchmarkovanie a volanie funkcií

jednotlivých algoritmov. Ostatné súbory zahrnuté v projekte pridávajú funkcionalitu rôznych hash funkcií a implementáciu API medzi programovacími jazykmi C++ a C.

Metriky, ktoré sú implementované v súbore algo_benchmarks.c, zahŕňajú vkladanie elementov, zmazanie polovice elementov, vyčistenie algoritmu, úspešné vyhľadávanie a neúspešné vyhľadávanie. Vkladanie elementov prebieha v cykle pre počet špecifikovaný v konfigurácii štruktúry Benchmark. Pre všetky ostatné metriky prvé jadro vykoná vloženie *N* elementov bez toho, aby bol zavolaný switch_perf_modeling. Po príprave a vložení elementov sa benchmark spustí s povoleným switch_perf_modeling. V prípade, že pri vkladaní elementov nastane chyba, prvé jadro uvoľní alokované zdroje a program sa ukončí. Funkcie implementované v algo_benchmarks.c iba ďalej volajú API, uloženú v štruktúre Benchmark. Štrukúra alghorithm_interface je zložená z ukazovateľov na funkcie vybraného algoritmu, ktorý je práve benchmarkovaný. Táto API združuje implementácie viacerých algoritmov a zároveň umožňuje využitie algoritmov napísaných v programovacom jazyku C++ do programovacieho jazyka C.

6.1.1 Použitie C++ algoritmov v jazyku C

Program implementovaný v praktickej časti tejto práce je napísaný v programovacom jazyku C. Ako už bolo spomenuté, jazyk C bol vybraný kvôli jeho rýchlosti, jednoduchosti a kompatibilite s hardvérovým simulátorom RISC-V. Poskytuje najpresnejšiu simuláciu reálneho prostredia, na ktorom by sa benchmarkované algoritmy využívali. Taktiež je vhodný z toho dôvodu, že algoritmy napísané v C++ je možné jednoducho používať pomocou spoločného API súboru.

Väčšina voľne dostupných algoritmov, ktoré budú integrované do benchmarkovacieho programu, je implementovaná v programovacom jazyku C++. Tu nastáva problém, že programovacie jazyky C a C++ nie sú plne kompatibilné. Pre možnosť použitia algoritmov implementovaných v programovacom jazyku C++ v C programe je nutné implementovať API, ktorá bude predstavovať rozhranie medzi týmito jazykmi.

V programovacom jazyku C nie je možné vytvoriť inštanciu C++ triedy. Toto obmedzenie však vieme obísť jednoduchým spôsobom. Namiesto inicializácie algoritmu priamo v C súbore, môžeme vytvoriť C++ súbor, ktorý implementuje funkciu pre inicializáciu mapy. Takéto C++ rozhranie nám spolu so štruktúrou ukazovateľov umožní pracovať s C++ algoritmami v programovacom jazyku C.

Každý voľne dostupný algoritmus je uložený v pod zložke s názvom „third-party“. V zložke každého algoritmu je taktiež C++ súbor, ktorý vytvára rozhranie medzi C++ algoritmom tretej strany a C kódom benchmarkovacieho programu. Pre každý algoritmus tretej strany je implementované také rozhranie, ktoré napĺňa štruktúru alghorithm_interface funkciami potrebnými k benchmarku. Toto rozhranie taktiež spája implementáciu rôznych algoritmov, ktoré napríklad môžu potrebovať vykonať viacero úkonov pre inicializáciu algoritmu. S algoritmom sa v hlavnej časti programu pracuje ako s void ukazovateľom, kde rozhranie algoritmu pretypuje void ukazovateľ na ukazovateľ konkrétneho algoritmu.

6.1.2 Integrácia hash funkcií

Pre autentickosť benchmarku je potrebné používať kľúče a hodnoty, ktoré predstavujú reálne prípady použitia vysoko rýchlostných algoritmov v Ethernete. To znamená, že kľúče a hodnoty uložené a používané algoritmom môžu byť uložené v rôznych dátových typoch.

Pri bežných operáciách, ktoré vyžadujú využitie hash funkcie, sa pracuje s kľúčom. Dĺžka ukladanej hodnoty a jej dátový typ je teda irelevantný pre integráciu hash funkcií. Kľúč v kontexte Ethernetu môže reprezentovať viaceré hodnoty ako napríklad VLAN-ID, IPv4 adresu alebo IPv6 adresu. VLAN-ID je 12 bitová hodnota, ktorá môže byť jednoducho reprezentovaná v 16 bitovej celočíselnej hodnote bez znamienka dátovým typom uint16_t, ktorý je definovaný v štandardnom hlavičkovom súbore stdint.h. IPv4 adresa má presne 32 bitov. Na uloženie IPv4 je vhodný 32 bitový celočíselný dátový typ bez znamienka uint32_t, ktorý je definovaný v rovnakom hlavičkovom súbore ako uint16_t. Adresa IPv6 sa skladá zo 128 bitov a jej dátový typ nie je tak jednoznačný ako pri predošlých dvoch príkladoch. Pre tento prípad je vhodné využiť pole bytov, ktoré môžeme v programovacom jazyku C inicializovať ako pole ukazovateľov na uint8_t hodnoty.

Pre celočíselné dátové typy a pre pole bytov sa využívajú rozdielne hash funkcie. Celočíselné dátové typy môžu využívať std::hash funkciu, ktorá je implementovaná ako základ pre väčšinu voľne dostupných algoritmov použitých v tejto práci. Std::hash používa hash funkciu v závislosti na dátovom type kľúča. Pre celočíselné hodnoty využíva std::hash identity hash funkciu, ktorá jednoducho vráti číslo odpovedajúce vstupnej hodnote kľúča.

Z oficiálnej C++ dokumentácie: „Pre reťazce znakov programovacieho jazyka C neexistuje žiadna špecializácia. std::hash<const char*> vytvára hash hodnoty ukazovateľa (z adresy pamäte), neskúma obsah poľa znakov.“ Pre pole znakov vracia std::hash iba hash

adresy ukazovateľa na prvý znak. Toto ju robí nepoužiteľnou pre vypočítanie hashu spomínanej IPv6 adresy. V tomto prípade je nutné použiť inú hash funkciu, ktorá dokáže pracovať s poľom znakov. Medzi takéto funkcie patria napríklad voľne dostupné hash funkcie MurmurHash3 a Wyhash, ktoré počítajú hash na základe celého poľa znakov, a tým garantujú, že sa pre istú IPv6 hodnotu vygeneruje vždy rovnaká hash hodnota. [34]

6.2 Implementácia static_hash_map

Static_hash_map je implementácia hash tabuľky s otvoreným adresovaním. Na riešenie konfliktov sa v hash tabuľke používa robin hood hashovanie s lineárnym probovaním. Napriek jej pôvodnému menu sa v nej nepoužíva statické alokovanie pamäte, ale dynamické. Táto zmena bola vykonaná kvôli potrebám benchmarkovania. Statická alokácia môže byť stále použitá pri menšom množstve predom známych elementov. Toto značne urýchli niektoré funkcie hash tabuľky, obzvlášť vkladanie. Bez ohľadu na spôsob alokácie sa kapacita hash mapy vždy rozšíri na ďalšiu mocninu dvojky. Toto rozšírenie nám umožní používať rýchle modulo, ktoré bolo popísané v teoretickej časti tejto práce.

Štruktúra samotnej hash mapy drží informácie o jej kapacite, momentálnej veľkosti, maximálnom koeficiente zaťaženia, veľkosti kľúča, hodnoty a samotných párov kľúčov a hodnôt, ktoré sú uložené v poli. Jeden pár sa skladá z kľúča, hodnoty a 8 bitovej PSL hodnoty. PSL je špeciálna hodnota, ktorá nám pomáha určiť, či je pár v tabuľke „bohatý“ alebo „chudobný“. Pôvodná hodnota PSL je nastavená na 0. Ak je pár kvôli hashovej kolízii umiestnený do ďalšieho miesta v poradí, tak sa hodnota PSL zväčší o 1.

Funkcia map_init alokuje miesto potrebné na používanie hash mapy. Každý pár v hash tabuľke má hodnotu nastavenú na NULL a kľúč nastavený na NULL alebo 0 podľa dátového typu kľúča. Táto funkcia vracia void ukazovateľ na vytvorenú inštanciu hash mapy. Na začiatku funkcie sa skontroluje hodnota maximálneho koeficientu zaťaženia, ktorá sa nemôže rovnať alebo presahovať hodnotu 1.

Vkladanie do hash mapy sa vykonáva pomocou funkcie map_insert, ktorá skontroluje vstupy a následne vypočíta ideálny index pre vkladaný kľúč podľa rýchlej modulo operácie jeho hashu. Ak je miesto na ideálnom indexe v hash tabuľke voľné, tak sa pár jednoducho vloží do hash tabuľky na tento index a zväčší sa veľkosť hash mapy o jeden. V prípade, že na ideálnom indexe leží iný pár, tak sa ďalej postupuje podľa hodnoty PSL páru, ktorý na tomto indexe leží. Pokiaľ je rovnaký alebo väčší, tak sa lineárnym probovaním pokračuje na ďalší index v tabuľke a PSL hodnota páru, ktorú vkladáme do hash tabuľky, je zväčšená

o jedna. Toto sa opakuje, pokým nie je nájdené prázdne miesto v hash tabuľke alebo hodnota PSL vkladaného páru nie je nižšia ako hodnota vloženého páru. V tomto prípade si páry vymenia svoje úlohy a pôvodne vložený pár hľadá svoje nové miesto.

Mazanie z hash tabuľky prebieha podobne ako vkladanie. Taktiež sa skontrolujú vstupy a zavolá sa interná funkcia lookup, ktorá vráti index páru, ak v hash tabuľke existuje. Ak sa tento pár v tabuľke nenájde, tak funkcia map_erase hlási neúspech. Inak sa pár z tabuľky zmaže a ďalší pár v lineárnom poradí sa posunie na jeho miesto ak je jeho hodnota väčšia ako 0, t.j. neleží na svojom ideálnom indexe. Týmto docielime čo najkratší možný čas vyhľadávania, pretože páry budú ležať čo najbližšie k svojmu ideálnemu indexu.

Funkcia map_lookup jednoducho skontroluje vstupy a zavolá internú lookup funkciu. Táto lookup funkcia funguje podobne ako vyhľadávanie prázdneho miesta pri vkladaní páru. Avšak, nehľadá sa prázdne miesto ale konkrétny kľúč. Táto funkcia implementuje nekonečný cyklus, ktorý vráti hodnotu ak sa kľúč nájde alebo sa nájde prázdny kľúč. Keďže hodnota maximálneho koeficientu zaťaženia sa nemôže rovnať alebo presahovať konštantu 1, tak je zrejmé, že sa tento nekonečný cyklus vždy zastaví na prázdnom mieste v hash tabuľke, keďže hash tabuľka na základe tohto koeficientu nemôže byť nikdy plná na 100 percent alebo viac.

Map_clear vráti hash tabuľku do stavu tak, že zmaže každý pár v tabuľke. Táto funkcia neuvoľní žiadne miesto, pretože predpokladá, že hash tabuľka bude používaná naďalej s inými dátami.

Ďalšie dve funkcie, ktoré nie sú vkladané do štruktúry alghorithm_interface benchmarku, sú map_print a map_free. Funkcia map_print sa nepoužíva a je určená čisto na debugovacie účely. Funkcia map_free sa používa v hlavnom benchmarkovacom programe ako súčasť funkcie free_resources. Ak bola v benchmarku použitá static_flat_map tak sa zavolá táto funkcia aby sa uvoľnila dynamicky alokovaná pamäť.

6.3 Implementácia bloom_filter

Bloom_filter je implementáciou Bloom filtra, ktorý poskytuje základnú funkcionalitu inicializácie, vkladania a vyhľadávania v Bloom filtri. Navyše poskytuje funkcie pre vyčistenie Bloom filtra a uvoľnenie alokovaných zdrojov. Ako už bolo spomenuté v teoretickej časti, z Bloom filtra nie je možné elementy vymazať bez toho, aby sme celý Bloom filter neprestavali.

Funkcia bloom_init nám poskytuje rozhranie pre vytvorenie inštancie Bloom filtra. V tejto funkcii sa alokuje potrebná pamäť a podľa vstupných parametrov sa vypočíta ideálna veľkosť filtra v bitoch a počet hash funkcií, ktoré budú použité pre vypočítanie indexov v bitovom poli.

$$n = \frac{m * \log(r)}{\log(\frac{1}{2^{\log(2)}})}$$

Rovnica 5. Výpočet ideálnej bitovej veľkosti Bloom filtra

Rovnica 5. popisuje matematický vzťah použitý pre vypočítanie ideálnej bitovej veľkosti Bloom filtra *n* s predpokladaným množstvom elementov *m* a požadovanou mierou falošne pozitívnych výsledkov *r*. Bitová veľkosť je celočíselne zaokrúhlená nahor. Potom je pomocou bitových operácií nastavená na nasledujúcu mocninu dvojky. Toto je opäť implementované z dôvodu využitia rýchlej modulo operácie.

Bitové pole je implementované ako pole neznačených celočíselných hodnôt o veľkosti 4 bajtov – uint32_t. Jednotlivé bity sú v tomto poli nastavované pomocou bitových operácií OR a vyhľadávané pomocou bitových operácií AND.

$$k = \frac{n}{m} * \log(2)$$

Rovnica 6. Výpočet ideálneho počtu hash funkcií Bloom filtra

Výpočet ideálneho počtu hash funkcií *k* pre Bloom filter sa počíta pomocou rovnice 6., kde *n* značí bitovú veľkosť Bloom filtra a *m* označuje predpokladané množstvo elementov. Pre vyššie množstvo elementov by hľadanie *k* rozdielnych hash funkcií bolo príliš náročné. Jednoduché a efektívne riešenie je použitie jednej rovnakej hash funkcie pre každú hash operáciu *k* s tým, že k hashovanému číslu je pripočítaný index, ktorý zodpovedá poradiu hash funkcie z počtu *k*. Týmto dosiahneme *k* hash funkcií aj pri vyššom počte *k*.

Vkladanie je jednoduchá operácia, pri ktorej sa overia vstupy a do Bloom filtra sa vloží *k* hash bitov. Toto je dosiahnuté vypočítaním indexu pre každý *k* bit a nastavenie hodnoty v Bloom filtri na jednotku.

Pri vyhľadávaní sa rovnaký počet *k* bitov skontroluje. Ak sú všetky bity nastavené na jednotku, tak je vrátená jednotka. Tento stav značí, že daný element sa môže nachádzať v dátovej štruktúre, ktorá ho drží, a môžeme pokračovať vo vyhľadávaní. Ak je aspoň jeden

z *k* bitov nastavený na nulu, tak je možné tvrdiť, že sa element určite v dátovej štruktúre, ktorá ho drží, nenachádza a vyhľadávanie je okamžite ukončené.

Funkcia bloom_clear vynuluje všetky bity v bitovom poli. Táto operácia je vhodná pre privedenie Bloom filtra do pôvodného stavu po inicializácii, čo šetrí čas, ktorý by bol inak spotrebovaný pre alokáciu pamäte pre nový Bloom filter.

Nakoniec funkcia bloom_free uvoľní alokovanú pamäť Bloom filtra. Volanie tejto funkcie je vhodné v prípade, ak Bloom filter v aplikácií naďalej nebude používaný alebo potrebujeme zvýšiť množstvo alokovanej pamäte pre vloženie viacerých elementov.

6.4 Implementácia externého skriptu

Hardvérový simulátor RISC-V akcelerátoru poskytuje vyhodnotenia behu programu vo viacerých oblastiach ako napríklad celkový beh v nanosekundách, percento cache miss v rôznych úrovniach pamäte cache a iné. Najzávažnejšie obmedzenie simulátoru je jeho neschopnosť tieto údaje zbierať počas behu programu. Z tohto dôvodu vzniká potreba vytvorenia externého skriptu, ktorý bude program spúšťať s rôznymi nastaveniami konfigurácie.

Ako už bolo spomenuté, skript benchmark.py je napísaný v programovacom jazyku Python. Jeho úlohou je spustiť benchmark mnohonásobne, s rôznymi nastaveniami konfigurácie. Za každé spustenie sú z výsledku simulátoru extrahované informácie, ktoré sú vykreslené do grafov. Jeho výška sa odvíja od hodnoty nameranej a extrahovanej pomocou hardvérového simulátoru. Grafy sa postupne ukladajú do zložky s názvom graphs. Táto zložka obsahuje pod zložky, ktoré sú pomenované podľa toho, či pri benchmarku bol použitý Bloom filter alebo nie. V každej z týchto zložiek nájdeme pod zložky pre hash funkcie, ktoré boli použité pre benchmark. Hash zložka taktiež obsahuje viacero pod zložiek pre celkový čas benchmarku, L1D cache miss rate, L2 cache miss rate. Tieto zložky obsahujú samotné grafy pomenované podľa metrík, ktoré popisujú. Po ukončení behu pre jednu konfiguráciu je vytvorený Excel súbor, ktorý obsahuje výsledky. Po dobehnutí všetkých konfigurácií sa vytvoria Excel súbory s celkovým poradím algoritmov.

Pre vykresľovanie grafov je v skripte využívaný modul matplotlib. Matplotlib.pyplot poskytuje jednoduchú manipuláciu a vytváranie rôznych grafov. Na opätovné spúšťanie benchmarkového programu sa používa modul subprocess, ktorý nám umožňuje spustiť benchmarkovací program s rôznymi konfiguráciami. Na spracovanie dát do Excel tabuliek sú použité moduly pandas spolu s openpyxl.

7 BENCHMARK

V tejto časti práce budú porovnané algoritmy na hardvérovom simulátore RISC-V akcelerátoru od firmy NXP Semiconductors. Porovnanie bude prevedené vo forme grafov a výsledných Excel tabuliek.

Pre pokrytie čo najvyššieho počtu rôznych prípadov bolo spustených veľké množstvo benchmarkovacích konfigurácií, ktoré sa líšia v parametroch, akými sú: veľkosť kľúča, veľkosť uloženej hodnoty, maximálny koeficient zaťaženia, počet jadier použitých súčasne a rozsah množstva hodnôt s ktorými sa pracovalo.

Keďže z týchto konfigurácií vzniklo veľké množstvo benchmarkov, nebude každý graf a tabuľka vložená a popísaná v tejto práci, avšak budú popísané niektoré vybrané konfigurácie a ich výsledky. Všetky Excel tabuľky a grafy, ktoré sú výsledkom besnchmarku, budú priložené k tejto práci ako príloha.

V tejto časti práce budú vyhodnotené a popísané namerané dáta pre rôzne prípady na základe konfigurácií. V závere tejto práce bude odporučený algoritmus, ktorý je všeobecne najvhodnejší. Táto kapitola sa však bude venovať aj konkrétnejším prípadom a bude obsahovať odporúčania pre tieto prípady.

7.1 Konfigurácia benchmarku

Ako bolo spomenuté v predošlej časti, benchmark kvôli robustnosti zahŕňa veľké množstvo konfigurácií. Tieto konfigurácie sa snažia čo najbližšie priblížiť a simulovať prostredie Ethernetu.

Počet jadier, ktoré súbežne pracujú na benchmarku, bol rozdelený na 1, 4, 8, 16, 32 a 64 RISC-V jadier. Keďže RISC-V simulátor nepodporuje súčasnú operáciu zapísania dát do pamäte, boli viacjadrové konfigurácie využité len pre vyhľadávacie operácie, ktoré nevyžadujú súčasný zápis jadier.

Veľkosť kľúča a veľkosť dát bola vybraná tak, aby reprezentovala údaje, ktoré nesú Ethernet pakety. Pre kľúč boli vygenerované celočíselné čísla bez znamienka o veľkostiach 2, 4 a 16 bajtov reprezentujúce VLAN-ID, IPv4 adresu a IPv6 adresu. Pre hodnoty boli vygenerované celočíselné čísla bez znamienka pre 64, 128 a 256 bajtov, ktoré reprezentujú rôzne veľkosti Ethernet paketov.

Keďže sa v praktickej časti tejto práce testovali hash tabuľky, bolo potrebné otestovať ich výkon pre rôzne maximálne koeficienty zaťaženia. Hodnoty maximálneho koeficientu zaťaženia boli nastavené na 30, 50 a 90 percent pre test hash tabuliek pri nízkom (30%), strednom (50%) a vysokom (90%) koeficiente zaťaženia.

Ďalší faktor, ktorý bolo potrebné zmerať, bolo zaťaženie v rámci množstva elementov, s ktorými algoritmus pracuje. Tieto množstvá boli zvolené na rozsahy nižší a vyšší. Nižší rozsah obsahuje 200 – 1 000 elementov s tým, že boli otestované množstvá o veľkosti násobku 200 (200, 400, 600, atď.). Vyšší rozsah podobným štýlom obsahuje 2 000 – 10 000 elementov s krokom 2 000.

Posledným nastavením konfigurácie bolo taktiež využitie Bloom filtru. Využitie Bloom filtru môže zrýchliť neúspešné vyhľadávanie za nízke režijné náklady na pamäť a tým vzniká potreba odmerať výkon hash tabuliek s využitím Bloom filtra a bez neho.

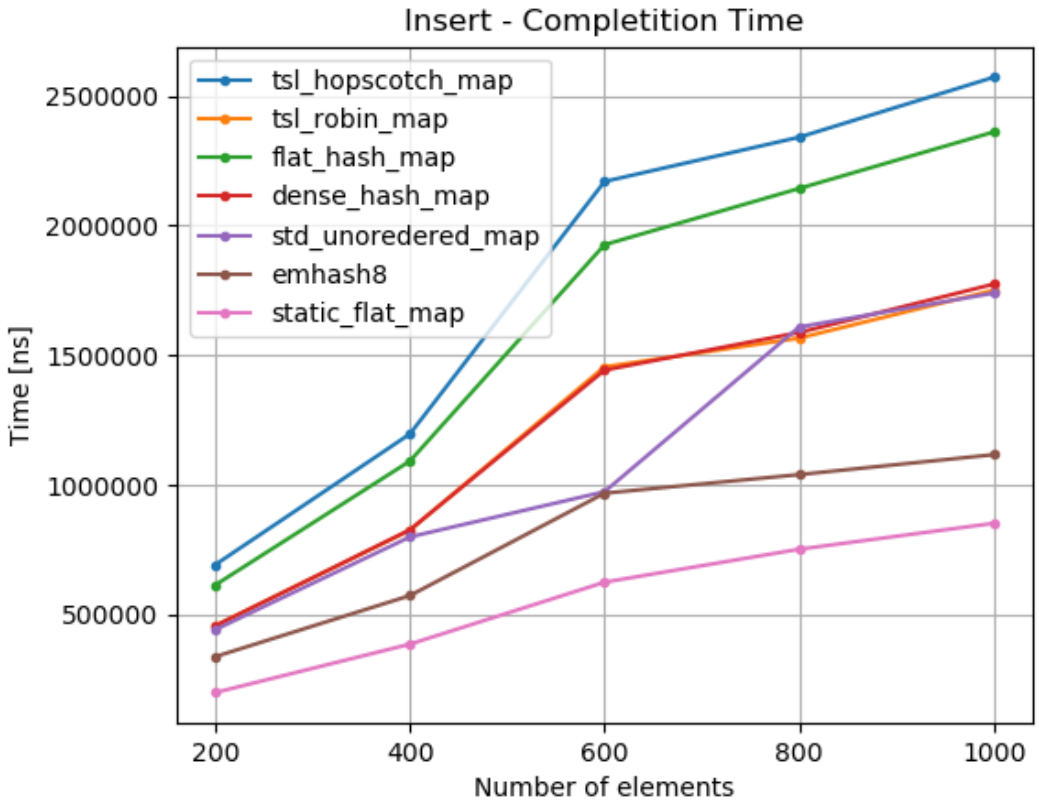
7.2 Popis vybraného benchmarku

V tejto časti budú popísané samotné výsledky benchmarku pre vybranú konfiguráciu, a taktiež grafy a Excel tabuľky, ktoré vznikli ako výsledok benchmarkovacieho programu vypracovaného v tejto práci.

Vygenerované výsledky pre konkrétnu konfiguráciu môžeme nájsť v prílohe risc-v_benchmakrs.zip. Pod zložka data obsahuje všetky vygenerované Excel súbory, zatiaľ čo položka graphs obsahuje všetky grafy. Konkrétne benchmarky sa označujú napríklad: „E200-1000-K4-V128-LF50-C1", kde „E" popisuje počet elementov, „K" veľkosť kľúča v bajtoch, „V" veľkosť hodnoty v bajtoch, „LF" percento maximálneho koeficientu zaťaženia a „C" vyjadruje počet použitých RISC-V jadier. V pod zložke data je taktiež možné nájsť dva súbory nazvané Rankings-200-1000.xlsx a Rankings-2000-10000.xlsx. Tieto súbory obsahujú výsledné hodnotenia pre rôzne kombinácie parametrov veľkosti kľúča a počet súbežných jadier, ktoré budú použité pri vypracovaní nasledujúcej kapitoly. Konfigurácia, ktorá bude hlbšie popísaná, využíva nasledujúce nastavenia:

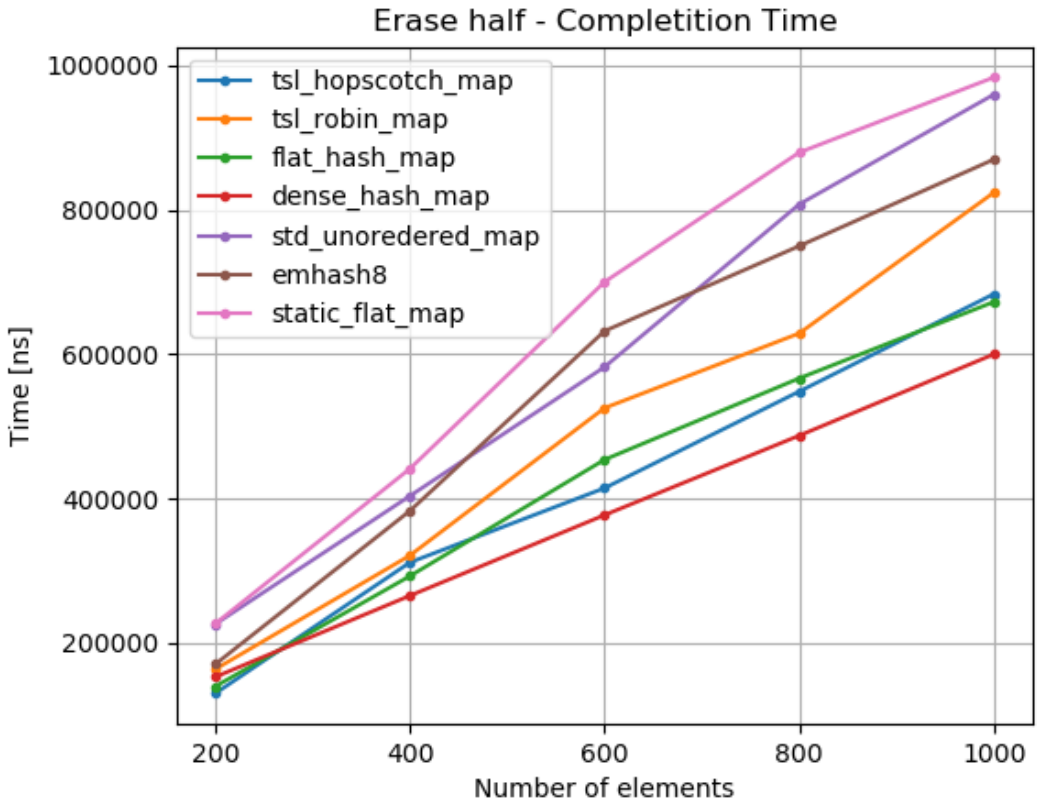
- veľkosť kľúča: 4B
- veľkosť hodnoty: 128B
- maximálny koeficient zaťaženia: 50%
- počet súbežných jadier: 1

Vygenerované výsledky pre konfiguráciu je možné nájsť pod názvom „E200-1000-K4-V128-LF50-C1“. Grafy, ktoré sú v tejto kapitole práce zvolené na ukážku, boli zvolené na základe algoritmu, ktorý sa umiestnil na prvom mieste. Benchmarky sú vytvorené aj pre L1 a L2 cache miss rate, ale v tejto časti budú popísané hlavne s ohľadom na čas behu benchmarkovacieho programu. Pre rýchlosť algoritmu v kontexte Ethernetu budú najdôležitejšie hlavne metriky týkajúce sa vyhľadávania. Avšak je potrebné zvážiť aj rýchlosť vkladania a mazania.



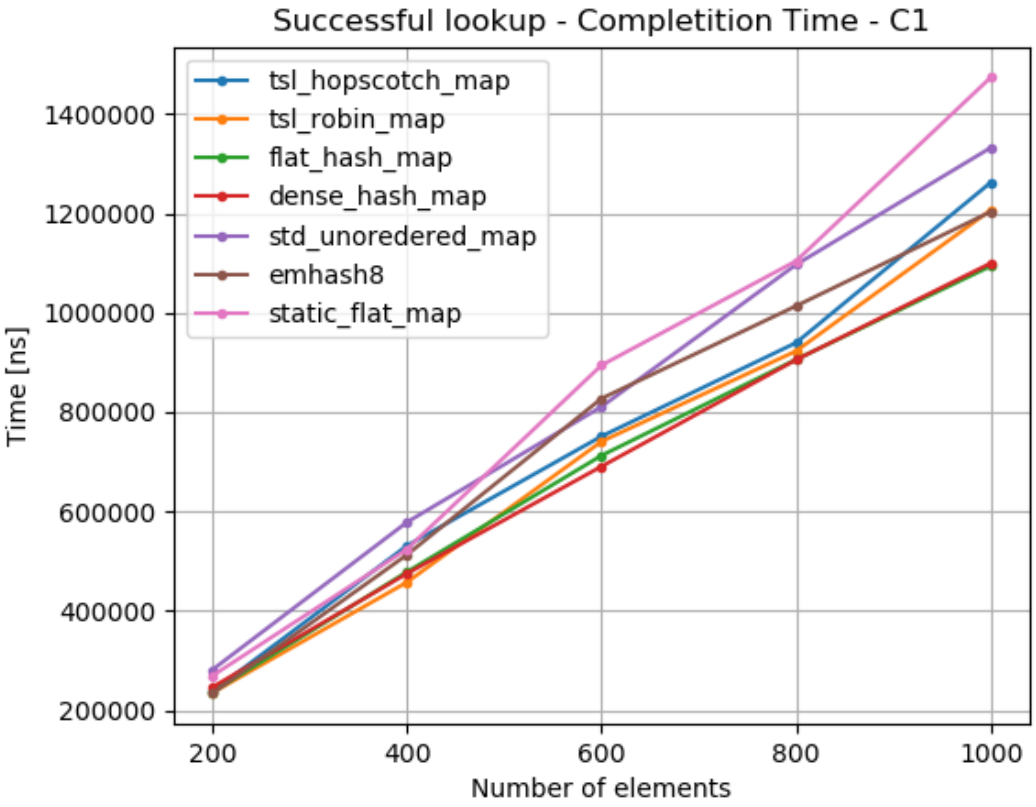
Obrázok 8. Graf Insert/std_hash/no_bloom

Graf zobrazený na obrázku 8. reprezentuje vkladanie elementov do rôznych hash tabuliek, ktoré sú zobrazené v legende grafu. Os X reprezentuje počty elementov od 200 po 1000 a os Y reprezentuje čas v nanosekundách, ktorý uplynul pre vybrané počty elementov. Z grafu je evidentné, že static_flat_map bol najrýchlejší pri tejto konfigurácií benchmarku. V tomto grafe môžeme taktiež vidieť, že tsl_hopsotch_map bol najpomalší zo všetkých hash máp vo vkladaní pri spomínanej konfigurácií.



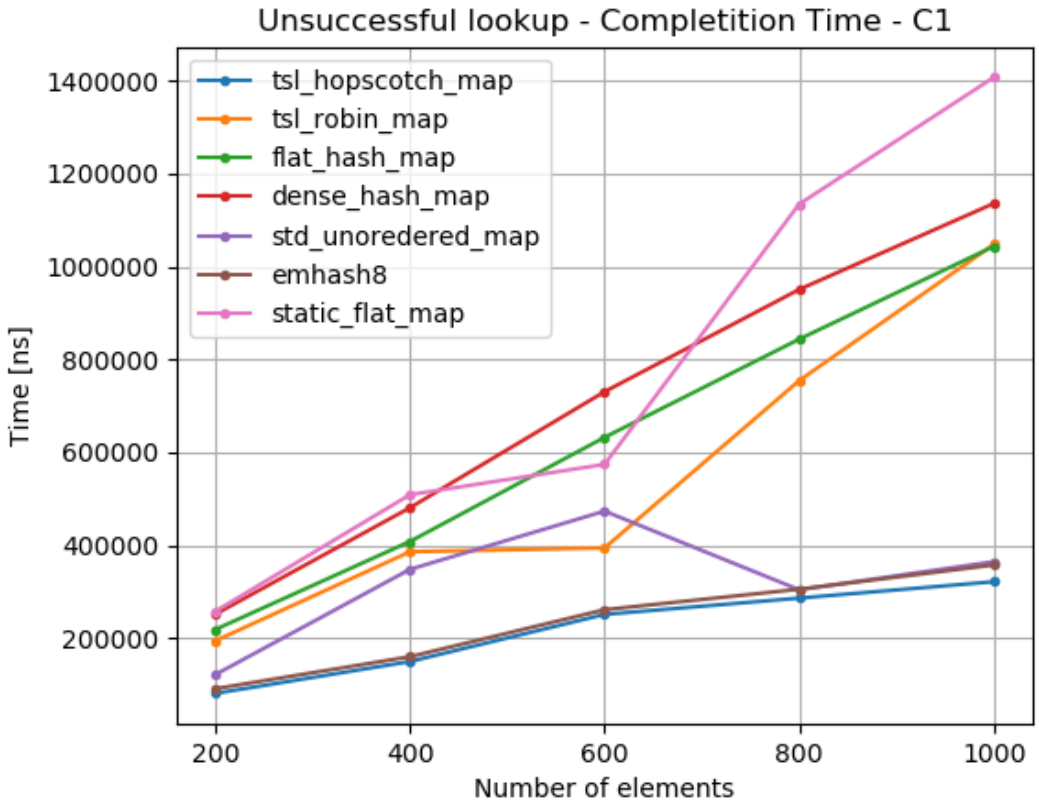
Obrázok 9. Graf Erase half/magic_hash/no_bloom

Na obrázku 9. je zobrazený graf „Erase half“, ktorý zobrazuje metriku, pri ktorej je zmazaná z hash tabuľky polovica elementov na porovnanie rýchlosti operácie mazania. Z grafu je zrejmé, že dense_hash_map bol najrýchlejší algoritmus pri meraní tejto metriky. Z grafu taktiež môžeme zhodnotiť fakt, že výkon hash máp všetkých algoritmov, okrem static_flat_map, má v tejto kategórii pri veľmi nízkom počte elementov malý rozptyl.



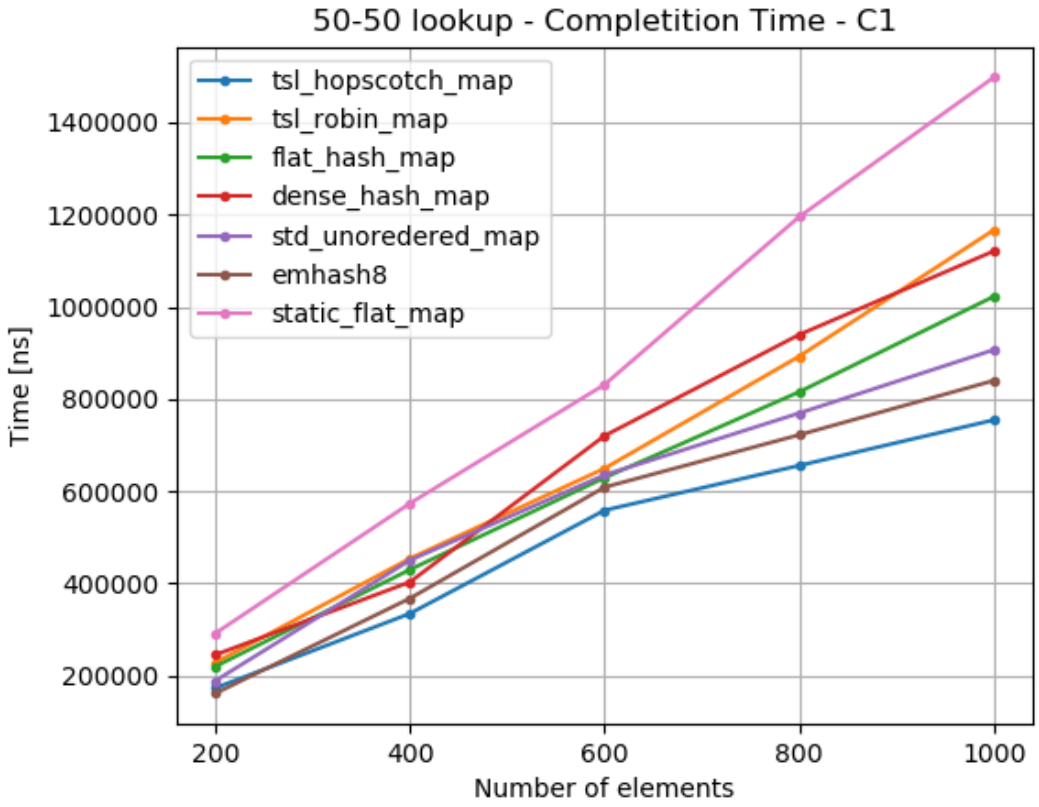
Obrázok 10. Graf Successful lookup/magic_hash/no_bloom

Na grafe z obrázku 10. môžeme vidieť, že metrika „Successful lookup“ je veľmi konkurencieschopná. Najlepšie výsledky dosiahli hash mapy flat_hash_map a dense_hash_map. Z grafu nie je zjavné, ktorá z týchto máp bola najlepšia, ale v Excel dokumente môžeme vidieť, že hash mapa dense_hash_map bola v tomto benchmarku najvýkonnejšia.



Obrázok 11. Graf Unsuccessful lookup/magic_hash/no_bloom

Z grafu metriky „Unsuccessful lookup“ zobrazeného na obrázku 11. je možné vidieť, že hash tabuľky emhash8 a tsl_hopscotch_map dosahujú najlepšie časy. Tsl_hopscotctch_map dosahuje najvyšší výkon pre túto metriku. Za zmienku stojí aj to, že std_unordered_map je nečakane výkonná pri vyhľadávaní elementov, ktoré nie sú v tabuľke, v porovnaní s predošlou „Successful lookup“ metrikou.



Obrázok 12. Graf 50-50 lookup/magic_hash/no_bloom

Posledný graf, ktorý bude v tejto práci popísaný, je možné vidieť na obrázku 12., ktorý zobrazuje metriku „50-50 lookup“. Pri tejto metrike je zaistené, že polovica operácií vyhľadávania zlyhá a druhá polovica uspeje. Z grafu je zrejmé, že najvýkonnejší algoritmus pri tejto metrike bol tsl_hopscotch_map. Všetky ostatné algoritmy boli relatívne konkurencieschopné až na static_flat_map, ktorá bola jednoznačne pomalšia pri každom rozmere elementov.

Z pozorovaní na grafoch v tejto časti diplomovej práce môžeme vyvodiť nasledujúce výsledky. Pri zameraní na vybraný benchmark E200-1000-K4-V128-LF50-C1 môžem odporúčať vynechanie Bloom filtra, keďže sa ukazuje ako pomalšia možnosť dokonca aj pri metrike „Unsuccesful lookup“. Ako hash funkciu by som odporúčal magic_hash, poprípade std_hash ako alternatívu. Ostatné hash funkcie využité v tomto benchmarku sa využívajú na hashovanie reťazcov, čo je obecne pomalšie ako jednoduchý hash vypočítaný z čísla. Hash tabuľku by som odporučil podľa jej použitia. V rámci Ethernetu musíme klásť vyšší dôraz na vyhľadávacie operácie. S predpokladom, že väčšina vyhľadávacích operácií bude úspešná, by som odporúčal použiť dense_hash_map pre jej rýchlosť v metrike „Successful

lookup“. Pri viac pesimistickom predpoklade, by som uprednostnil tsl_hopscotch_map pre jej rýchlosť v zmiešanom a neúspešnom vyhľadávaní.

7.3 Výsledky benchmarkov

V tejto kapitole budú preskúmané Excel súbory, ktoré zhŕňajú výsledky benchmarkov. Excel súbory je možné nájsť v prílohe 1. v pod zložke „data“. Ich názvy sú Rankings200-1000.xlsx a Rankings2000-10000.xlsx. V každom súbore sú výsledky rozdelené do hárkov v závislosti na počte súbežných jadier a veľkosti kľúča. Každý Excel hárok obsahuje päť stĺpcov. V prvom stĺpci sú názvy porovnávaných hash máp a informácia o tom, či bol využitý Bloom filter pre danú mapu. Ďalšie štyri stĺpce obsahujú poradie algortimov v závislosti na čase, L1/L2 cache miss rate a celkové poradie, ktoré berie do úvahy každú metriku. V tejto práci bude najväčší dôraz kladený na čas, keďže v kontexte Ethernetu je to najpodstatnejšia metrika.

Použitie správnej hash funkcie je dôležité pre výkon hash tabuľky. Z výsledných hodnotení v „Rankings“ Excel súborov je možné tvrdiť, že hash funkcie std_hash a MurmurHash3 podporujú dosiahnutie čo najrýchlejšieho vyhľadávania. Pre hashovanie celočíselných hodnôt by som odporúčal použiť std_hash, ktorý používa jednoduchú identity hash funkciu. MurmurHash3 by som odporúčal pre hash tabuľky, ktoré používajú polia bajtov ako svoje kľúče. Magic_hash je dobrá alternatíva k hash funkcií std_hash, ktorá je pri niektorých 64 jadrových benchmarkoch dokonca rýchlejšia.

Použitie Bloom filtra by teoreticky malo zrýchliť vyhľadávanie v prípade, že veľké množstvo vyhľadávaní je neúspešných. V praxi je ale z výsledných benchmarkov možné vidieť, že toto funguje iba pre malé množstvo vložených elementov. Je možné, že to je spôsobené sub-optimálnou implementáciou a odporúčal by som ešte hlbšie preskúmanie možností s použitím Bloom filtra. Pre vyhľadávanie v Ethernete na zvolenom RISC-V akcelerátore od firmy NXP Semiconductors by som neodporúčal použitie Bloom filtrov.

Názov hash mapy	Počet umiestnení na prvom mieste
emhash8	32
tsl_robin_map	4

Tabuľka 2. Počet výhier hash máp v benchmarku

Tabuľka 2. zobrazuje počet umiestnení na prvom mieste v benchmarku, pre každý algoritmus, ktorý vyhral aspoň jeden krát. Z informácii, ktoré nám poskytuje tabuľka 2., je zrejmé, že hash mapa emhash8 vyhrala benchmark vo väčšine konfigurácií. Hash mapa tsl_robin_map niekedy zvíťazila pri 16 bajtových kľúčoch, kde sa používa hashovanie podľa bajtov a hodnota je ukladaná ako pole znakov.

Na základe prvenstva hash mapy emhash8 v mnohých benchmarkoch by som odporúčal používať práve emhash8 ako predvolenú na RISC-V Ethernet akcelerátore od firmy NXP Semiconductors, a taktiež z dôvodu, že je stále aktívne aktualizovaná.

Pre konkrétny prípad konfigurácie, kde sa prevažne používajú 16 bajtové kľúče, a pre vyhľadávanie, kde sa používa všetkých 64 RISC-V jadier, by som odporúčal zvážiť aj tsl_robin_map ako alternatívu k emhash8.

Mapa flat_hash_map sa zvyčajne dostáva do prvej trojice najrýchlejších algoritmov, avšak túto mapu by som neodporúčal používať, pretože bola vyvinutá jedným človekom a už nie je pravidelne aktualizovaná. Pre konkrétne konfigurácie by som odporúčal spoľahnúť sa na konkrétne benchmarky a ich výsledky podľa rýchlosti.

Hash mapu static_flat_map by som neodporúčal používať v produkcii nikdy, pretože má veľmi pomalé vyhľadávanie v porovnaní s ostatnými hash mapami.

ZÁVER

Cieľom diplomovej práce bolo vybrať vhodný vysoko výkonný algoritmus pre vyhľadávanie v Ethernete na embedded RISC-V akcelerátore od firmy NXP Semiconductors.

V rámci praktickej časti diplomovej práce, bol vytvorený benchmarkovací program, ktorý je schopný porovnávať voľne dostupné hash tabuľky na simulátore spomínaného RISC-V akcelerátoru. Zároveň bola vytvorená vlastná hash tabuľka, ktorá slúži ako základ pre benchmarkovanie ostatných voľne dostupných algoritmov a Bloom filter, ktorý mal potenciál zrýchliť neúspešné vyhľadávania.

Výsledok tejto práce je voľne dostupný benchmarkovací program pre embedded systémy, napísaný v programovacom jazyku C. Ďalším výsledkom práce, je odporúčanie vhodného algoritmu pre Ethernet vyhľadávanie s použitím RISC-V akcelerátoru NXP Semiconductors.

Ako predvolenú hash mapu pre zvolený RISC-V Ethernet akcelerátor od firmy NXP Semiconductors by som odporúčal emhash8, pretože je najrýchlejšia vo veľkom množstve konfigurácií, je pravidelne aktualizovaná a je obsiahnutá v jednom hlavičkovom súbore, čo znamená jednoduchú integráciu. Túto hash mapu by som odporúčal používať bez použitia Bloom filtra. Ako hash funkciu odporúčam std_hash pre hashovanie celočíselných kľúčov a MurmurHash3 pre hashovanie bajtových reťazcov.

Praktická časť tejto práce by s dostatkom času mohla byť rozšírená o ďalšiu funkcionalitu, kde by sa k porovnaniu viacerých hash funkcií a algoritmov pre vyhľadávanie v Ethernete pridalo viac možností na výber vhodného algoritmu. Ďalšia možnosť rozšírenia praktickej časti je preskúmanie možností použitia Bloom filtra v Ethernet vyhľadávaní, čo by bolo možné dosiahnuť optimalizáciou implementovaného Bloom filtra alebo využitím voľne dostupného Bloom filtra na porovnanie.

Ako ďalšia možnosť rozšírenia by mohla byť aj implementácia funkcií, ktoré sú dostupné ako ukazovatele na funkcie v štruktúrach alghorithm_interface a alghorithm_intance. Tieto funkcie by nám pomohli lepšie pracovať s benchmarkovaním a použitím voľne dostupných algortimov pre využitie vo vysoko výkonnom Ethernet vyhľadávaní.

ZOZNAM POUŽITEJ LITERATÚRY

[1] BANZAL, Prof. Shashi. *Data and Computer Network Communication*. Third Edition. Laxmi Publications, 2022. ISBN 9789393738462.

[2] E. SPURGEON, Charles. *Ethernet: The Definitive Guide*. O'Reilly Media, 2000. ISBN 9781565926608.

[3] IBM. *Ethernet networks*. Online. 2023. Dostupné z: https://www.ibm.com/docs/en/i/7.5?topic=standards-ethernet-networks. [cit. 2024-01-03].

[4] OSHANA, Robert a KRAELING, Mark. *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*. Second Edition. Elsevier Science and Technology Books, 2019. ISBN 9780128094488.

[5] IXIA. *Automotive Ethernet: An Overview*. Online. 2014. Dostupné z: https://support.ixiacom.com/sites/default/files/resources/whitepaper/ixia-automotive-ethernet-primer-whitepaper_1.pdf. [cit. 2024-03-04].

[6] LAVOIE, Sarah. *The Ultimate Guide to PoE*. Online. 2023. Dostupné z: https://www.onlogic.com/company/io-hub/what-is-poe-power-over-ethernet/. [cit. 2024-03-04].

[7] ASHJAEI, Mohammad; LO BELLO, Lucia; DANESHTALAB, Masoud; PATTI, Gaetano; SAPONARA, Sergio et al. *Time-Sensitive Networking in automotive embedded systems: State of the art and research opportunities*. Online. 2021. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1383762121001028. [cit. 2024-01-18].

[8] *Introduction to Precision Time Protocol (PTP)*. Online. 2024. Dostupné z: https://networklessons.com/cisco/ccnp-encor-350-401/introduction-to-precision-time-protocol-ptp. [cit. 2024-01-18].

[9] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. Online. 2. IEEE, 2008. ISBN 978-0-7381-5400-8. Dostupné z: https://ieeexplore.ieee.org/document/4579760. [cit. 2024-01-18].

[10] *IEEE Standard for Local and metropolitan area networks--Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP)*. Online. 6. IEEE, 2010.

ISBN 978-0-7381-6501-1. Dostupné z: https://ieeexplore.ieee.org/document/5594972. [cit. 2024-01-18].

[11] STALLINGS, William. *Computer Organization and Architecture*. 10. Pearson, 2015. ISBN 0134101618.

[12] *Hash Tables*. Online. 2022. Dostupné z: https://programming.guide/hash-tables.html. [cit. 2024-01-03].

[13] P. MEHTA, Dinesh a SAHNI, Sartaj. *Handbook of Data Structures and Applications*. Second Edition. CRC Press, 2018. ISBN 9781498701853.

[14] YELURI, Sharada. *Longest Prefix Matching in Networking Chips*. Online. 2023. Dostupné z: https://community.juniper.net/blogs/sharada-yeluri/2023/01/02/longest-prefix-matching-in-networking-chips. [cit. 2024-01-04].

[15] HOLM, Mark. *TCAM Demystified*. Online. 2020. Dostupné z: https://learningnetwork.cisco.com/s/article/tcam-demystified. [cit. 2024-01-03].

[16] SAGGURTI, Prasad. *Introduction to TCAM*. Online. 2023. Dostupné z: https://www.synopsys.com/designware-ip/technical-bulletin/introduction-to-tcam.html. [cit. 2024-01-04].

[17] *Comprehensive C++ Hashmap Benchmarks*. Online. 2022. Dostupné z: https://martin.ankerl.com/2022/08/27/hashmap-bench-01/. [cit. 2024-05-05].

[18] *Std::unordered_map*. Online. 2023. Dostupné z: https://en.cppreference.com/w/cpp/container/unordered_map. [cit. 2024-02-26].

[19] SKARUPKE, Malte. *I Wrote The Fastest Hashtable*. Online. 2017. Dostupné z: https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/. [cit. 2024-02-26].

[20] GOOGLE. *Sparsehash*. Online. 2020. Dostupné z: https://github.com/sparsehash/sparsehash. [cit. 2024-02-26].

[21] META. *F14 Hash Table*. Online. 2018. Dostupné z: https://github.com/facebook/folly/blob/main/folly/container/F14.md. [cit. 2024-02-26].

[22] META. *Open-sourcing F14 for faster, more memory-efficient hash tables*. Online. 2019. Dostupné z: https://engineering.fb.com/2019/04/25/developer-tools/f14/. [cit. 2024-02-26].

[23] TESSIL. *A C++ implementation of a fast hash map and hash set using robin hood hashing*. Online. 2022. Dostupné z: https://github.com/Tessil/robin-map. [cit. 2024-02-26].

[24] TESSIL. *A C++ implementation of a fast hash map and hash set using hopscotch hashing*. Online. 2022. Dostupné z: https://github.com/Tessil/hopscotch-map. [cit. 2024-02-26].

[25] *Fast and memory efficient open addressing c++ flat hash table/map*. Online. 2021. Dostupné z: https://github.com/ktprime/emhash. [cit. 2024-02-26].

[26] *Libcuckoo*. Online. 2018. Dostupné z: https://github.com/efficient/libcuckoo. [cit. 2024-02-26].

[27] *Segmentation Offloads*. Online. 2024. Dostupné z: https://docs.kernel.org/networking/segmentation-offloads.html. [cit. 2024-02-20].

[28] CONOLE, Aaron a LEITNER, Marcelo Ricardo. *Hardware Accelerator*. Online. 2024. Dostupné z: https://www.cadence.com/en_US/home/explore/hardware-accelerators.html. [cit. 2024-02-19].

[29] NXP SEMICONDUCTORS. *SJA1110 TSN ETHERNET SWITCH*. Online. 2022. Dostupné z: https://www.nxp.com/docs/en/fact-sheet/SJA1110AUTESFS.pdf. [cit. 2024-02-19].

[30] NXP SEMICONDUCTORS. *S32G PFE Product Brief*. Online. Rev. 2.0. 2023. Dostupné z: https://www.nxp.com/docs/en/product-brief/S32G2PFEPB.pdf. [cit. 2024-02-19].

[31] *The RISC-V Instruction Set Manual*. Volume I: Unprivileged ISA. 2019.

[32] SYNOPSYS. *What is RISC-V?* Online. 2022. Dostupné z: https://www.synopsys.com/glossary/what-is-risc-v.html#a. [cit. 2024-02-19].

[33] DREPPER, Ulrich. *What every programmer should know about memory*. Online. 2007. Dostupné z: https://lwn.net/Articles/252125/. [cit. 2024-02-13].

[34] Std::hash. Online. 2024. Dostupné z: https://en.cppreference.com/w/cpp/utility/hash. [cit. 2024-03-12].

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

API Application Programming Interface

AVB Audio Video Bridging

BCAM Binary Content Addressable Memory

CAM Content Addressable Memory

CPU Central Processing Unit

CSMA/CD Carrier Sense Multiple Access with Collision Detection

FIB Forwarding Information Base

GPGPU General-Purpose Graphics Processing Unit

IEEE Institute of Electrical and Electronics Engineers

IP Internet Protocol

ISA Instruction Set Architecture

LAN Local Area Network

LLC Logical Link Control

LPM Longest Prefix Match

MAC Media Access Control

MMRP Multiple MAC Registration Protocol

MSRP Multiple Stream Registration Protocol

MVRP Multiple VLAN Registration Protocol

NIC Network Interface Controller

PFE Packet Forwarding Engine

PoE Power over Ethernet

PSL Probe Sequence Lengths

PTP Precision Time Protocol

RAM Random Access Memory

RISC Reduced Instruction Set Computer

SIMD Single Instruction, Multiple Data

SRAM Static RAM

SRP Stream Reservation Protocol

TCAM Ternary Content Addressable Memory

TCP Transmission Control Protocol

TSN Time Sensitive Networking

VLAN Virtual LAN

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM PRÍLOH

Príloha 1. riscv_benchmarks.zip