Improving performance in software internet routers through compact lookup structures and efficient datapaths

Zec, Marko

Doctoral thesis / Disertacija

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:455203

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-29**

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Marko Zec

IMPROVING PERFORMANCE IN SOFTWARE INTERNET ROUTERS THROUGH COMPACT LOOKUP STRUCTURES AND EFFICIENT DATAPATHS

DOCTORAL THESIS

Supervisors:
Associate Professor Miljenko Mikuc, PhD
Professor Luigi Rizzo, PhD

Zagreb, 2019

Marko Zec

POBOLJŠANJE IZVEDBE PROGRAMSKIH INTERNETSKIH USMJERITELJA POMOĆU KOMPAKTNIH PREGLEDNIH STRUKTURA I EFIKASNIH PODATKOVNIH STAZA

DOKTORSKI RAD

Mentori:
izv. prof. dr. sc. Miljenko Mikuc
prof. dr. sc. Luigi Rizzo

Zagreb, 2019

Doktorski rad izrađen je na Sveučilištu u Zagrebu, Fakultetu elektrotehnike i računarstva, na Zavodu za telekomunikacije.

Mentori:
izv. prof. dr. sc. Miljenko Mikuc
prof. dr. sc. Luigi Rizzo

Doktorski rad ima: 77 stranica

Doktorski rad br.: _____

Mentor's Curriculum Vitae

Miljenko Mikuc is an Associate Professor at the University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Telecommunications. He graduated in 1987. and received his PhD in the field of technical sciences, electrical engineering in 1997. from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER).

He has participated on seven scientific projects of the Ministry of Science, Education and Sports of the Republic of Croatia and on the international project "Verification and validation methods for formal descriptions (COST 247). He participated and led projects with "The Boeing Company - IDS, LabNet Analysis, Modeling Simulation and Experimentation", "International Computer Science Institute / University of California, Berkeley", "The FreeBSD Foundation", multi-year research project in the field of information and communication technology with Ericsson Nikola Tesla d.d., "Customized IMUNES for Ericsson (E-IMUNES)" and IRI-project "New Generation Lawful Interception System - NG LI", where FER is a partner of the company SedamIT d.o.o.

At the University of Zagreb, Faculty of Electrical Engineering and Computing, he is responsible for the following undergraduate and graduate courses: "Digital Logic", "Network Programming", "Internet Security" and "Network and Services Management." At the postgraduate doctoral study program, he is responsible for "Formalisms in Telecommunications" and "Communication Protocols - Selected Topics". As a mentor, he successfully led more than 200 undergraduate and graduate students, while at the postgraduate study he was a mentor for 15 master theses and 3 doctoral theses.

He has published over 40 scientific and professional papers in journals and conference proceedings in the field of communication networks, protocols, virtualization, formal methods and security. He is a member of a professional association of IEEE. He participates in the work of the Technical Program Committee of the International Scientific Conference of SoftCOM and as a reviewer at a number of international conferences.

Životopis mentora

Miljenko Mikuc je izvanredni profesor na Zavodu za telekomunikacije Fakulteta elektro-tehnike i računarstva Sveučilišta u Zagrebu. Diplomirao je 1987. godine a doktorsku disertaciju iz područja tehničkih znanosti, polja elektrotehnika, "Postupci provjere ispravnosti specifikacije telekomunikacijskih procesa" obranio je 1997. godine.

Sudjelovao je kao istraživač na sedam znanstvenih projekata Ministarstva znanosti, obrazovanja i sporta Republike Hrvatske te na međunarodnom projektu "Verification and validation methods for formal descriptions (COST 247)". Bio je voditelj dva projekta Primjene informa-

cijske tehnologije pod pokroviteljstvom Ministarstva znanosti i tehnologije Republike Hrvatske. Bio je voditelj projekata suradnje s "The Boeing Company - IDS, LabNet Analysis, Modeling Simulation and Experimentation", "International Computer Science Institute", "The FreeBSD Foundation" iz SAD-a te višegodišnjeg istraživačkog projekta u sklopu suradnje na području informacijskih i komunikacijskih tehnologija s kompanijom Ericsson Nikola Tesla d.d., "Prilagođen IMUNES za Ericsson (E-IMUNES)". Voditelj je IRI-projekta "Nova generacija rješenja za zakonsko presretanje podataka - NG LI" na kojem je FER partner tvrtke prijavitelja SedamIT d.o.o.

Sudjeluje u nastavi na preddiplomskom i diplomskom studiju kao nositelj ili su-nositelj predmeta "Digitalna logika", "Mrežno programiranje", "Sigurnost u Internetu" i "Upravljanje mrežom i uslugama". Na poslijediplomskom studiju su-nositelj je predmeta "Formalizmi u telekomunikacijama" i "Odabrana poglavlja komunikacijskih protokola". Pod njegovim mentorstvom uspješno je završilo studij više od 200 studenata na preddiplomskom i diplomskom studiju. Na poslijediplomskom studiju bio je mentor pri izradi 15 magistarskih radova te 3 doktorska rada.

Objavio je preko 40 znanstvenih i stručnih radova u časopisima i zbornicima konferencija u području komunikacijskih mreža, protokola, virtualizacije, formalnih metoda i sigurnosti. Član je stručne udruge IEEE. Sudjeluje u radu tehničkog programskog odbora međunarodne znanstvene konferencije SoftCOM, te kao recenzent na većem broju međunarodnih konferencija.

Mentor's Curriculum Vitae

Luigi Rizzo is a Professor of Computer Engineering at the Universita' di Pisa, Italy. His research focuses on computer networks and operating systems. He has published over 50 academic papers, including highly cited works on network emulation, scalable reliable multicast and multicast congestion control, packet scheduling, high speed network I/O and virtual machine networking. His research projects received funding from the European Commission, as well as many industrial partners including Microsoft, Cisco, Intel, Netapp, Verisign.

Much of his work has been implemented and deployed in popular operating systems and applications, and widely used by the research community. His contributions include the popular dummynet network emulator (a standard component of FreeBSD and OS/X, and now also available for Linux and Windows); one of the first publicly available erasure code for reliable multicast; the qfq packet scheduler; and the netmap framework for fast packet I/O.

Prof. Rizzo has been a visiting researcher at several industrial and academic research institutions, including ICSI (UC Berkeley), Intel Research Cambridge (UK), Intel Research Berkeley, and recently Google Mountain View. He has served as General Chair for SIGCOMM 2006, TPC Co-Chair for SIGCOMM 2009 and CoNEXT 2014, and TPC member/reviewer for a number networking conferences and journals.

Životopis mentora

Luigi Rizzo je profesor računalnog inženjerstva na Sveučilištu Pisa, Italija. Glavna područja njegovih istraživanja su računalne mreže i operacijski sustavi. Objavio je preko 50 radova koji su citirani više od 8000 puta, uključujući radove o emulaciji računalnih mreža, skalabilnog pouzdanog višeodredišnog odašiljanja i kontrole zagušenja višeodredišnog odašiljanja, raspoređivanja paketa, brzog dohvaćanja i odašiljanja paketa, te umrežavanja virtualnih strojeva. Njegovi istraživački projekti bili su financirani sredstvima Europske komisije, kao i brojnih industrijskih partnera, uključujući Microsoft, Cisco, Intel, Netapp, Verisign.

Velik dio njegovih radova implementiran je i ugrađen u popularne operacijske sustave i aplikacije, te široko prihvaćen u istraživačkoj zajednici. Njegovi doprinosi uključuju popularni alat za emulaciju mreža dummynet koji je standardna komponenta operacijskih sustava FreeBSD i OS/X, te je dostupan i za Linux i Windows OS; jedan od prvih javno dostupnih brišučih kodova za pouzdano višeodredišno odašiljanje; algoritam raspoređivanja paketa QFQ; te prilagodni sloj za brzo dohvaćanje i odašiljanje paketa netmap.

Prof. Rizzo bio je gostujući istraživač na nizu industrijskih i sveučilišnih istraživačkih institucija, uključujući (UC Berkeley), Intel Research Cambridge (UK), Intel Research Berkeley, te trenutno Google Mountain View. Predsjedao je znanstvenim skupom SIGCOMM 2006, su-

predsjedao tehničkim programskim odborom skupova SIGCOMM 2009, CoNEXT 2014, te je bio član programskih odbora ili recenzent niza časopisa i skupova iz područ računalnih mreža.

Acknowledgments

Prof. Miljenko Mikuc and prof. Luigi Rizzo were great and patient advisors, and I thank them and congratulate them both on finally graduating me. Jointly writing and polishing the original DXR paper, which set the foundation for this thesis, was among the most fun and rewarding experiences I recall in my whole career.

Most likely I would have never got this thesis complete without additional guidance from a shadow advisor, prof. Maja Matijašević. Her help was also instrumental in getting the more recent results documented and published in a decent form.

Denis Salopek set up our physical evaluation testbed and did the most of the benchmarking and profiling work on a prototype packet processor based on DXR, the results of which unfortunately could not be included in this thesis, but led to ironing out several subtle bugs in the DXR library and the Click element.

Most of all, I wish to thank my wife Nataša and our kids for being patient and supportive with me wasting time on stop-and-go efforts around this thesis over just too many years. Well, it's finally done!

Abstract

Expensive, inflexible, closed yet fast hardware packet datapath implementations have dominated the high-speed and core Internet routing scene over the past two decades due to both real and perceived lack of performance offered by software routers running on commodity hardware. Today, software routers are mostly displaced to edge functions where the throughput pressure is lower, or to applications where flexibility takes precedence over performance.

This thesis challenges the aforementioned status-quo by asserting that the performance potential of contemporary multi-core microprocessors for applications in Internet routing datapaths may be significantly greater than what is currently thought. The results suggest that significant improvements in software-based Internet routing performance may be achieved by carefully engineering data structures and algorithms to permit modern microprocessors to efficiently leverage their fast and sizable cache hierarchies, thereby extracting more parallelism across multiple execution cores, while preserving the precious main memory bandwidth for packet input and output, or other memory-intensive input / output tasks.

The research presented here focuses on longest prefix matching (LPM) as a fundamental operation which simultaneously presents a major performance bottleneck and implementation challenge in Internet Protocol (IP) routers, hardware and software based alike. The proposed algorithm and the accompanying data structures sustain nearly 3.5 billion random LPM lookups per second in a contemporary routing database containing 739,561 IPv4 prefixes with 148 unique next-hops, while running on conventional, commodity PC hardware. The same configuration can exceed 7 billion lookups per second with locality in the stream of lookup keys. The thesis dissects how the principles and techniques applied in the design and implementation of the experimental prototype contribute to achieving those throughput levels.

Keywords: IP lookups, LPM, software packet processors, software routers, router performance

Poboljšanje izvedbe programskih internetskih usmjeritelja pomoću kompaktnih preglednih struktura i efikasnih podatkovnih staza - prošireni sažetak

Tijekom proteklih dvadesetak godina programski ostvarene usmjeritelje temeljene na mikroprocesorskim platformama opće namjene u jezgri Interneta su u potpunosti istisnuli iz upotrebe visokopropusni ali skupi, zatvoreni, i nefleksibilni usmjeritelji temeljeni na specijaliziranim sklopovljima. Ovaj rad preispituje navedeni status-quo hipotezom da potencijal modernih višejezgrenih mikroprocesora može biti dostatan za učinkovitu primjenu u podatkovnim stazama brzih Internetskih usmjeritelja. Rezultati pokazuju da se povećanje propusnosti programski ostvarenih usmjeritelja može postići pažljivim odabirom i konstrukcijom podatkovnih struktura i algoritama koji modernim mikroprocesorima omogućuju dobro iskorištenje predmemorija, što se posredno odražava i učinkovitim paralelnim izvođenjem na više procesorskih jezgri uz malu učestalost pristupa glavnoj memoriji, čija sabirnica više vremena ostaje slobodna za prihvat i odašiljanje paketa, te za ostale memorijski zahtjevne ulazno / izlazne zadaće. Disertacija je usredotočena na problem pretraživanja tablica usmjeravanja najduljim prefiksnim podudaranjem (engl. longest prefix matching, LPM) kao temeljni postupak odlučivanja pri obradi paketa u Internetskim usmjeriteljima. Predložene nove klase algoritama s pripadajućim podatkovnim strukturama tijekom izvođenja na računalu opće namjene omogućuju postizanje propusnosti od približno 3.5 milijarde pretraživanja u sekundi temeljem nezavisnih, slučajno odabranih ključeva u tablici s 739.561 IPv4 zapisa i 148 moguća odredišta, preuzetoj iz usmjeritelja u jezgri Interneta. Uz pretraživanje s uzastopnim ponavljanjem slučajno odabranih ključeva postiže se propusnost do 7 milijardi upita u sekundi. U radu se analizira kako načela i postupci primjenjeni u oblikovanju i ostvarenju eksperimentalnog prototipa doprinose postizanju ovakvih razina propusnosti.

Rad je podijeljen u sedam poglavlja.

U prvom, uvodnom poglavlju, identificiran je istraživački problem, motivacija i ciljevi istraživanja te je dan opis strukture rada. Izložen je kratak pregled razvoja tehnologije usmjeritelja namjenjenih radu u jezgri Interneta. Opisana je tranzicija od široke primjene programski ostvarenih usmjeritelja tijekom devedesetih godina prošlog stoljeća, do njihovog kasnijeg potpunog napuštanja i zamjene specijaliziranim sklopovskim izvedbama. Rast propusnosti transmisijskih veza za nekoliko redova veličine, uz istodobni superlinearni rast broja mreža oglašenih u globalne tablice usmjeravanja, bili su glavni čimbenici nedostatne propusnosti usmjeritelja temeljenih na tadašnjim mikroprocesorskim platformama opće namjene. Tijekom 90-ih godina prošlog stoljeća predložen je niz algoritama za poboljšanje učinkovitosti programski ostvarenog pretraživanja globalnih tablica usmjeravanja, informacije o kojima Internetski usmjeritelji

razmjenjuju putem protokola Border Gateway Protocol (BGP). Tadašnje tablice usmjeravanja sadržavale su manje od 100.000, dok današnje premašuju 760.000 zapisa, pri čemu njihov broj i dalje nezaustavljivo raste, trenutno dinamikom od oko 50.000 zapisa godišnje. Već na prijelazu stoljeća do tada razvijeni algoritmi za programsko pretraživanje tablica usmjeravanja pokazali su se nedostatnim za praktičnu primjenu u opisanim uvjetima i postupno se zamijenjuju specijaliziranim sklopovljem u novijim generacijama Internetskih usmjernika visoke propusnosti. S druge strane, razvoj sve bržih specijaliziranih sklopovskih arhitektura (engl. application-specific integrated circuits, ASIC) za usmjeravanje paketa, zbog izrazito visokih troškova može pratiti svega nekoliko najvećih svjetskih proizvođača komunikacijske opreme. Male kompanije i akademske istraživačke grupe praktički nemaju izgleda za proboj i natjecanje u sve užem krugu takvih proizvođača, čime se koči razvoj inovacija. U praksi se pokazalo da, zbog svojih fiksnih kapaciteta, sklopovske arhitekture Internetskih usmjeritelja imaju kratak eksploatacijski rok trajanja, tipično od svega nekoliko godina. Problem je ilustriran primjerom iz kolovoza 2014. godine kad su zabilježene značajne oscilacije i nedostupnost pojedinih mreža u Internetu zbog rasta globalnih tablica usmjeravanja preko praga od 512 * 1024 IPv4 zapisa, što je bilo sklopovski uvjetovano ograničenje dijela tadašnjih usmjeritelja. Uz već spomenut visok rizik, dugotrajnost, i cijenu razvoja, povećanje kapaciteta obrade specijaliziranih sklopovskih usmjeritelja ograničeno je visokim razinama potrošnje i disipacije energije. Poseban problem je nefleksibilnost specijaliziranog sklopovlja u smislu prilagodbe modernim trendovima eksploatacije, kao što su virtualizacija mrežnih funkcija (engl. Network Function Virtualization, NFV), i programski definiranih mreža (engl. Software Defined Networking, SDN). Napredak tehnologije generičkih programirljivih logičkih sklopova (Field-Programmable Gate Array - FPGA) omogućio je njihovu praktičnu primjenu u platformama za obradu mrežnog prometa koje imaju bolje mogućnosti prilagodbe novim zahtjevima od usmjeritelja temeljenih na sklopovima ASIC, ali istodobno nude značajno manji kapacitet odnosno propusnost od sklopovskih rješenja. Upravo su spomenuta tehnološka ograničenja postojećih sklopovskih rješenja potaknula autora na istraživanje mogućnosti (ponovne) primjene modernih mikroprocesorskih platformi za brzu obradu paketa u Internetskim usmjeriteljima. U prvom poglavlju se opisuje i trenutni raskorak između zahtjeva za propusnošću današnjih transmisijskih tehnologija (10, 100, 400 Gbit/s) s kapacitetom obrade programski ostvarenih usmjeritelja temeljenih na operacijskim sustavima (OS) opće namjene. Rezultati vlastitih eksperimenata konzistentni su s izvješćima drugih autora koji ukazuju da je ograničenje propusnosti OS-a opće namjene reda veličine jednog milijuna paketa u sekundi (Mpps) po mikroprocesorskoj jezgri, što je dostatno tek za rad pri brzinama do 1 Gbit/s. Kao alternative tradicionalnom pristupu obradi paketa u jezgri OS-a citiraju se novije paradigme programske obrade mrežnog prometa, pri čemu niz autora predlaže i demonstrira značajno poboljšanje propusnosti zaobilaženjem mrežnog stoga OS-a prilikom prihvata i odašiljanja paketa. Međutim, rješenja problema učinkovitog pretraživanja

tablica usmjeravanja do nedavno se i dalje tražilo isključivo kroz delegiranje (engl. offload) na zasebno sklopovlje, npr. jezgre grafičkih procesora, ili raspoređivanjem prometa na više fizičkih računala opće namjene.

Drugo poglavlje ("Programska obrada paketa: sklopovska perspektiva") ispituje ključna svojstva suvremenih tržišno-dominantnih mikroprocesora: paralelizam i hijerarhije predmemorija. Dan je pregled stanja modernih višejezgrenih mikroprocesora opće namjene, gledano kroz prizmu zahtjeva za obradom velike količine međusobno nezavisnih podataka u jedinici vremena, što je svojstveno obradi mrežnog prometa u Internetskim usmjeriteljima, te posebno problemu brzog pretraživanja tablica usmjeravanja. Ističe se problem stagniranja rasta frekvencije radnog takta, koje su od početka 80-ih godina prošlog do prvih godina ovog stoljeća narasle za približno tri reda veličine, od četiri MHz do nešto manje od četiri GHz, što je kroz zadnjih petnaestak godina ostala gornja granica frekvencije radnog takta najbržih komercijalnih mikroprocesora. Istodobno je nastavljen tehnološki razvoj litografskih postupaka u proizvodnji poluvodiča, odnosno gustoće integracije elektroničkih elemenata, da bi i napredak tih tehnoloških procesa počeo pokazivati značajne trendove stagniranja tijekom posljednjih nekoliko godina. Kao temeljni čimbenici zaustavljanja napretka u brzini rada mikroprocesora ističu se povećanje propagacijskih kašnjenja unutar prospojnih puteva između elektroničkih elemenata pri novijim tehnološkim procesima, te posebno velika gustoća snage toplinske disipacije pri radu na visokim frekvencijama takta. Isti tehnološki problemi uvjetuju i teškoće u razvoju novih generacija specijaliziranih integriranih krugova za obradu i usmjeravanje Internetskog prometa, koji su dodatno ekonomski opterećeni visokim troškovima razvoja sklopovlja prolagođenog najnovijim tehnološkim procesima poluvodičke litografije, koje je teško amortizirati kroz relativno male serije u kojima se takvi specijalizirani sklopovi proizvode, za razliku od mikroprocesora opće namjene koji se proizvode masovno i imaju veliko, još uvijek nepresušno tržište. Tako je istaknut primjer dominantnog proizvođača komunikacijske opreme čiji se usmjeritelji najvećeg kapaciteta predviđeni za rad u jezgri Interneta i dalje temelje na specijaliziranom procesoru predstavljenom još 2013. godine, a čija je propusnost za današnje prilike skromnih 280 Mpps. U nastavku poglavlja analiziraju se temeljne značajke modernih mikroprocesorskih platformi opće namjene. To su mogućnost izvođenja više instrukcija u jednom ciklusu takta (engl. instruction-level parallelism / superscalar execution), dinamičko predviđanje grananja (engl. branch Ppediction), dinamički odabir i izvođenje instrukcija izvan programskog slijeda uz zadržavanje semantike slijednog izvođenja (engl. out-of-irder execution), višerazinske hijerarhije predmemorija (engl. caches) kojima se vrijeme slučajnog pristupa približno udvostručuje sa svakom razinom većeg kapaciteta, te glavne memorije velikog kapaciteta s vremenom slučajnog pristupa reda veličine do 100 ns, odnosno od oko 300 ciklusa radnog takta procesorske jezgre. Ispitivanje vremena slučajnog pristupa predmemorija i glavnoj memoriji provedeno je vlastitim jednostavnim programom pokrenutim na nizu računala izgrađenim oko različitih pro-

cesora proizvođača Intel i AMD, a opaženi rezultati konzistentni su s vrijednostima objavljenim od strane samih proizvođača. Uslijed stagnacije povećanja brzina izvođenja jedne programske dretve (engl. thread), moderni mikroprocesori sadrže sve više procesorskih jezgri koje dijele zajedničku glavnu memoriju i dio (zadnju razinu) predmemorije. Svaka procesorska jezgra uobičajeno podržava izvođenje do dvije nezavisne programske dretve, s glavnim ciljem iskorištenja sklopovlja u vrijeme dok druga programska dretva na istoj jezgri čeka na dohvat podataka iz predmemorije ili iz glavne memorije. Kroz konkretni eksperiment paralelnog pretraživanja tablica usmjeravanja prikazana je razlika u izvođenju algoritma na fizički odvojenim jezgrama u usporedbi s izvođenjem dvije dretve na jednoj jezgri, koje je manje učinkovito. Na kraju poglavlja iznosi se niz prijedloga za oblikovanje podatkovnih struktura i algoritama kao sinteza vlastitih i opažanja drugih autora, a koje proizlaze iz svojstava današnjih mikroprocesorskih platformi opće namjene. Zbog stagnacije napretka u brzini izvođenja pojedinačnih dretvi (engl. single-thread performance) što je inherentno svim generacijama mikroprocesora uvedenih na tržište tijekom proteklog desetljeća, ističe se nužnost fokusiranja na oblikovanje algoritama i podatkovnih struktura prilagođenih učinkovitom paralelnom izvođenju na više procesorskih jezgri. Zbog velikog raskoraka u brzini slučajnog pristupa predmemoriji i glavnoj memoriji podatkovne strukture nužno je oblikovati tako da zauzimaju što manje memorijskog prostora, kako bi čim većim dijelom i uz što manje istiskivanja (engl. spilling) mogle biti dohvaćane iz predmemorija procesora. Umjesto tradicionale raspršenosti manjih fragmenata podatkovnih struktura po širem adresnom prostoru, organiziranje podatkovnih struktura u kompaktne neprekinute blokove omogućuje konsolidaciju memorijskih stranica (engl. pages) uobičajene veličine od 4 KB u veće cjeline (engl. superpages) koje obuhvaćaju 1 MB do 4 MB linearnog adresnog prostora, što smanjuje potrebu za intervenciju operacijskog sustava u upravljanju sklopovskim translacijskim tablicama (engl. translation lookaside buffers - TLB) virtualne memorije. Korištenja pokazivača koji na današnjim 64-bitnim procesorskim arhitekturama imaju mali omjer korisne informacije i zauzeća prostora treba gdje je moguće zamijeniti indeksiranjem, primjena kojeg osim manjeg zauzeća memorije inherentno potiče organizaciju podataka u kompaktnije, linearne, neraspršene strukture. Podatke koje se većinu vremena čita a rijetko ažurira mogže se dijeliti između više programskih dretvi, ali strukture koje se često ažuriraju potrebno je alocirati u nezavisnim instancama za svaku dretvu, kako bi se smanjila potreba za implicitnim (sklopovskim) i eksplicitnim (programskim) sinkronizacijskim operacijama nad dijeljenim blokovima podataka. Dugo vrijeme slučajnog pristupa podacima u glavnoj memoriji može se dijelom kompenzirati korištenjem procesorskih instrukcija za najavu pristupa (prefetching) kako bi se podaci unaprijed dohvatili u predmemoriju za tijekom izvođenja instrukcija neovisnih o ciljanim podacima.

Treće poglavlje ("Direktno / rasponsko pretraživanje najdužih prefiksnih podudaranja") predstavlja temeljna načela koja se nalaze iza ključnog doprinosa, sheme pretraživanja nazvane

Direct / Range, skraćeno DXR. U uvodnom dijelu poglavlja opisuju se glavne značajke popularnog IPv4 LPM postupka DIR-24-8, razvijenog krajem prošlog stoljeća s ciljem (tada) učinkovite sklopovske izvedbe uz korištenje dediciranih memorijskih modula DRAM, a danas je u širokoj primjeni kao standardni modul odnosno biblioteka u programskim platformama za obradu mrežnog prometa Click i DPDK. Postupak DIR-24-8 se oslanja na dvije pregledne tablice, od kojih se glavna (veća) formira projiciranjem IPv4 prefixa iz izvorne tablice s mrežnim maskama širine do 24 bita na linearno polje veličine 2^{24} elemenata. Elementi glavne pregledne tablice sadrže informaciju o usmjeravanju (engl. next hop, NH), ili u slučaju da u izvornoj tablici postoje zapisi s identičnih 24 bita veće težine ali s mrežnom maskom koja zahvaća više od 24 bita, element glavne pokazuje na segment druge (pomoćne) tablice u kojem se indeksiranjem s preostalih 8 bitova traženog ključa pronalazi konačna informacija o usmjeravanju. Opisana podjela odabrana je temeljem razdiobe širina mrežne maske (engl. prefix length) IPv4 zapisa u tablicama usmjeravanja u jezgri Interneta, u kojima su najzastupljenije mrežne maske širine do uključivo 24 bita, dok su širine mrežne maske od 25 bitova i više razmjerno rijetke. LPM pretraživanje u postupku DIR-24-8 je trivijalno, a svodi se na direktno indeksiranje glavne pregledne tablice pomoću 24 bita veće težine traženog IPv4 ključa, te po potrebi dodatnim indeksiranjem pomoćne tablice preostalim bitovima ključa. Međutim, kako pregledne tablice postupka DIR-24-8 svojom veličinom nadilaze kapacitet predmemorija većine mikroprocesora opće namjene, uz upite ključevima disperziranima po cijelom IPv4 adresnom spektru predmemorije gube na učinkovitosti, što je temeljni nedostatak postupka pri paralelnom izvođenju na višejezgrenim mikroprocesorima s dijeljenom glavnom memorijom. U nastavku poglavlja opisuje se koncept transformiranja tablica mrežnih prefixa u uređen slijed susjednih adresnih raspona (engl. ranges), koji obuhvaćaju cijeli IPv4 adresni prostor. Nakon takve transformacije LPM pretraživanje svodi se na trivijalno binarno pretraživanje adresnih raspona. Kako je svaki raspon definiran početnom i završnom 32-bitnom adresom te NH oznakom za koju se u praksi pokazalo da je dovoljno 16 bitova, za svaki zapis o adresnom rasponu dovoljno je 10 byteova, uz pretpostavku da se zapisi pohranjuju slijedno u kontinuirano polje kako bi se isto moglo iteracijski pretraživati. U ovakvoj podatkovnoj strukturi značajan je višak informacije, budući da je početna adresa svakog adresnog raspona uvijek jednaka završnoj adresi prethodnog uvećanoj za jedan. Izostavljanjem završne adrese raspona iz zapisa, koja se može izvesti iz početne adrese slijedećeg, veličina zapisa smanjuje se na 6 byteova. Veličina tablice s adresnim rasponima u najgorem slučaju proporcionalna je broju zapisa u izvornoj tablici, što bi za današnje izvorne tablice usmjeravanja s preko 750.000 IPv4 mreža rezultiralo tablicom raspona s do 1.5 milijun elemenata, ukupne veličine do 9 MByte, za pretraživanje koje bi bilo nužno do 20 iteracijskih koraka, što je neostvarivo u ciljanim vremenskim okvirima od nekoliko desetaka ciklusa procesorskog takta. Zauzeće memorije može se smanjiti podjelom adresnog prostora na 2^K jednakih blokova, te korištenjem početnih K bitova tražene adrese za direktno

indeksiranje dodatno uvedene pregledne tablice, pri čemu se adresni rasponi razvijaju zasebno za svaki od uniformnih blokova. Uz odabir vrijednosti K veći ili jednak 16, 16 bitova veće težine u zapisima s početnim adresama adresnih raspona postaju suvišni, pa se veličina pojedinog zapisa smanjuje s 6 na 4 bytea. Dodatno optimiranje veličine zapisa može se ostvariti za adresne raspone koji ne proizlaze iz IPv4 mreža s mrežnom maskom širom od 24 bita, te ukoliko mogućih odredišta (next hop) nema više od 256, u kojem slučaju nije potrebno pamtiti 8 bitova najmanje težine početne adrese raspona (uvijek su nula), pa je zauzeće memorije po zapisu moguće sažeti na svega 2 bytea. U nastavku poglavlja opisan je postupak iteracijskog pretraživanja s konkretnom izvedbom u programskom jeziku C. Postupak pretraživanja analizira se iz perspektive organizacije predmemorija modernih mikroprocesora u tzv. retke (engl. lines) veličine 64 bytea, što uz spomenutu linearnu organizaciju zapisa o adresnim rasponima osigurava da se zapisi pri završnim koracima iteracijskog postupka pretraživanja dohvaćaju iz razine predmemorije koja je najbliža procesorskoj jezgri te ima najmanje trajanje dohvata podataka. Slijedi opis i analiza postupka formiranja i ažuriranja preglednih tablica uz korištenje tablice usmjeravanja preuzete iz usmjernika u jezgri Interneta. Kao izvorišna baza (tablica) za pohranu informacija o IPv4 mrežama koristi se već postojeća, provjerena implementacija binarnog stabla sa skraćenim putevima preuzeta iz operacijskog sustava FreeBSD, na temelju koje se formiraju direktna tablica (engl. direct table) i tablica adresnih raspona (engl. range table). Zavisno od odabira vrijednosti parametra K, u rasponu između 16 i 20, postižu se ukupne veličine preglednih tablica između jednog i pet MByte, odnosno od 1.76 do 7.32 bytea po IPv4 mreži u izvorišnoj tablici. Odabirom veće vrijednosti parametra K smanjuje se broj iteracija potrebnih za razrješenje LPM upita, ali se povećava zauzeće memorije te vrijeme potrebno za formiranje preglednih tablica. Analizom tablice adresnih raspona za K veći ili jednak 16, uočena je pojavnost raspona koji imaju identične bitove manje težine i pripadajuće informacije o usmjeravanju, ali su povezani s odvojenim dijelovi IPv4 adresnog prostora, odnosno imaju različite prvih K bitova veće težine. Kako se K bitova veće težine ne pohranjuje u zapisima o adresnim rasponima, nego se razrješuju indeksiranjem prve tablice, opisane identične adresne raspone moguće je objediniti, i time smanjiti zauzeće memorije, što se pokazalo posebno učinkovito pri odabiru parametra K većih od 19. U nastavku poglavlja prikazani su i analizirani rezultati ispitivanja propusnosti algoritma s nizovima slučajno odabranih ključeva. Ispitivana su tri tipska scenarija za različite izvorišne tablice preuzete s usmjernika iz jezgre Interneta, s različitim konfiguracijama parametra K, na nizu različitih računala temeljenim na komercijalno dostupnim mikroprocesorima proizvođača AMD i Intel. U prvom tipskom scenariju svi su slučajno odabrani ispitni ključevi međusobno nezavisni. U drugom tipskom scenariju u ispitnu proceduru umjetno se uvodi međuzavisnost između uzastopnih pretraživanja superponiranjem rezultata prethodnog pretraživanja sa slijedećim slučajno odabranim ključem, čime se u značajnoj mjeri blokira mogućnost mikroprocesorske jezgre za špekulacijskim izvođenjem. U trećem

tipskom scenariju za svaki od slučajno odabranih ključeva LPM upit se ponavlja osam puta, s ciljem simuliranja pojave usnopljenosti mrežnih tokova. Rezultati eksperimenata pokazuju da se za prvi tip ispitivanja s nezavisnim slučajno odabranim ključevima najviša propusnost na jednoj procesorskoj jezgri, razina do 230 milijuna upita u sekundi (engl. million lookups per second - Mlps), postiže pri odabiru parametra K između 19 i 22, zavisno od mikroprocesora na kojem se eksperiment provodi, odnosno veličine njegovih predmemorija. U drugom tipskom ispitnom scenariju, s međuzavisnošću između slijednih LPM upita, postižu se propusnosti do 70 Mlps na jednoj procesorskoj jezgri. Pri paralelnom izvođenju algoritma na više procesorskih jezgri postižu se propusnosti do 2490 Mlps na 16-jezgrenom procesoru AMD Ryzen 7-1700. Eksperimenti pokazuju približno linearan rast propusnosti algoritma s raspoređivanjem na više procesorskih jezgri, dok isti eksperiment proveden s algoritmom DIR-24-8 pokazuje stagnaciju i pad ukupne propusnosti pri raspoređivanju na više od šest jezgri, u kojim uvjetima se postiže propusnost od ukupno 430 Mlps, dakle skoro red veličine manje od optimalne konfiguracije algoritma DXR na istom stroju. Poglavlje završava analizom učestalosti promašaja pri dohvatu podataka iz predmemorije (engl. cache miss) temeljem očitanja sklopovskih brojila takvih događaja u mikroprocesoru. Konkretno, za mikroprocesor Intel i7-4771 koji je opremljen s 8 MByte predmemorije zadnje razine (engl. last level cache), broj promašaja pri dohvatu iz predmemorije konstantan je za sve ispitivane konfiguracije algoritma do uključno $K = 21$, za koju je pregledne tablice zauzimaju ukupno 8.35 MByte. Prikazana je usporedna analiza učestalosti promašaja dohvata iz predmemorije za odabrane konfiguracije algoritma DXR i DIR-24-8 u režimu paralelnog izvođenja na više jezgri, iz kojih je vidljiva značajno veća učestalost promašaja pri izvođenju LPM pretraživanja algoritmom DIR-24-8, koja doseže razinu od 0.8 do 1 promašaja po LPM pretrazi, zavisno od kapaciteta predmemorije procesora na kojem se provodi ispitivanje. Za optimalno odabrane konfiguracije algoritma DXR bilježe se razine od 0.1 promašaja po LPM pretrazi, koje su neizbježno uvjetovane dohvatom unaprijed pripremljenih slučajno odabranih ključeva iz zasebne tablice. U opisanim mjerenjima promašaja dohvata iz predmemorije zabilježena je propusnost LPM pretraga algoritma DXR u rasponu od približno 1.5 do 5 puta većoj od propusnosti algoritma DIR-24-8.

Četvrto poglavlje ("Daljnje vremenske i prostorne optimizacije") opisuje i analizira optimizacije algoritma DXR koje se postižu uvođenjem dodatne tablice, kojom se direktno indeksiranje temeljem prvih K bitova ključa rastavlja u dva koraka. Razdvojene tablice nazvane su direktna (engl. Direct) i proširena (engl. eXtension) tablica. Direktna tablica indeksira se s početnih D bitova ključa, dok se s dodatnih X bitova indeksira odabrani blok u proširenoj tablici, pri čemu je zbroj $D + X = K$. Uvođenjem druge, proširene tablice, omogućuje se smanjenje zauzeća memorije, pronalaženjem i objedinjavanjem (engl. deduplication) blokova koji pokazuju na iste zapise u tablici adresnih raspona. Postupak objedinjavanja blokova proširene tablice pokazao se posebno učinkovit pri odabiru parametara $D = 16$, te $X > 3$. Konkretno, za

konfiguraciju $D = 16, X = 6$ postiže se sažimanje memorije od približno 70% u usporedbi s odgovarajućom konfiguracijom $K = 22$ temeljne inačice algoritma. Gledano iz druge perspektive, uz približno isto zauzeće memorije, inačica algoritma s dvije tablice moći će razriješiti do dva bita ključa više indeksiranjem tablica, prije prelaska na sporiji, iteracijski postupak pretrage tablice adresnih raspona. Skraćenje iteracijskog binarnog pretraživanja odražava se na povećanje propusnosti koja pri izvođenju na jednoj procesorskoj jezgti dosežu do 40% više razine u usporedbi s temeljnom inačicom. Pri paralelnom izvođenju optimiranog algoritma u konfiguraciji $D = 16, X = 6$ na svim logičkim jezgrama procesora AMD Ryzen-7 1700 postiže se propusnost od 3204 Mlps, u usporedbi s 2490 Mlps koje na istom računalu ostvaruje konfiguracija $K = 20$ temeljne inačice. Najviša ukupna propusnost od 3491 Mlps ostvarena je pri ispitivanju algoritma na procesoru AMD ThreadRipper 1950X. Spomenute razine propusnosti odnose se na prvi tipski test, uz nezavisne slučajno odabrane ključeve, dok se uz ponavljajuće upite za isti ključ (treći tipski test) na procesoru AMD 1950X postiže ukupna propusnost od 7207 Mlps. Važno svojstvo optimirane inačice algoritma je i značajno manje zauzeće memorije kod tablica usmjeravanja s malim brojem zapisa u usporedbi s temeljnom inačicom. Primjerice, mala tablica usmjeravanja sa samo pet IPv4 mreža i konfiguraciji $D = 12, X = 9$ preslikat će se u pregledne tablice sa zauzećem memorije od svega 15 Kbyte.

Peto poglavlje ("Integracija u podatkovne staze") opisuje mogućnosti integriranja algoritma DXR u programski ostvarene IPv4 usmjeritelje. Obrazlažu se razlozi odustajanja od početnih napora za ugradnju algoritma DXR u jezgru operacijskog sustava FreeBSD, te odluke da se algoritam implementira u dvije inačice nezavisne od OS-a. Inačica na kojoj su se provela sva ispitivanja opisana u ovom radu implementirana je kao *lookup* modul u popularnoj programskoj platformi za obradu paketa *Click*, pri čemu je najveći dio modula ostvaren u programskom jeziku C++, uz enkapsuliranje komponenti preuzetih iz operacijskog sustava FreeBSD (*radixtree*) u zasebnu klasu. Na temelju te referentne implementacije razvijena je i ispitana samostalna DXR biblioteka ostvarena u ANSI C-u, s ciljem lakše integracije u aplikacije neovisne o platformi *Click*. Korištenjem spomenute biblioteke i platforme za brzo dohvaćanje i odašiljanje paketa *netmap* konstruirana je ispitna aplikacija koja je pokazala mogućnost prosljeđivanja paketa uz provođenje LPM pretraga bez gubitaka pri brzini od 10 Gbit/s, odnosno 14.88 Mpps, korištenjem samo jedne mikroprocesorske jezgre. Ispitivanja rada pri višim brzinama prijenosa nije bilo moguće provesti tijekom izrade ovog rada uslijed nedostatka odgovarajuće opreme, odnosno mrežnih kartica.

Šesto poglavlje ("Pregled literature") daje kritički osvrt na stanje istraživanja u području teme doktorskog rada i objavljene rezultate drugih istraživača, s naglaskom na evoluciju LPM algoritma namijenjenih izvođenju na mikroprocesorskim platformama opće namjene, nakon čega slijede završne napomene u sedmom poglavlju.

Zaključno, znanstveni doprinos disertacije uključuje novu klasu algoritama i kompaktnih

podatkovnih struktura za brzo programsko pretraživanje usmjerivačkih informacija za protokol IPv4; poboljšanja učinkovitosti podatkovnih staza u programskoj komutaciji paketa boljim iskorištavanjem prostornih i vremenskih mogućnosti paralelne obrade na procesorima opće namjene; te izvedbu dvije parametrizirane verzije sheme pretraživanja DXR, kao programske biblioteke i kao komponente za korištenje u modularnom programskom usmjeritelju Click, uz empirijsku provjeru ispravnosti njihovoga rada.

Ključne riječi: internetski usmjeritelji, pretraživanje najdužih prefiksnih podudaranja, programska komutacija paketa

Contents

Chapter 1

Introduction

1.1 Background and motivation

In the early 21st century, the availability of ubiquitous, affordable, reasonably fast and reliable packet-switched communication on a global scale, in a network known as the Internet, became simply taken for granted, just like running water, electrical energy distribution, or mass transportation, became infrastructural norms over the course of the previous century. The entire human society is becoming intrinsically dependent on packet-switched communication.

The increases in global Internet traffic volume [1], the pressure on the global routing system [2] [3] [4], and particularly the advances in transmission link speeds, have been among the major driving forces behind the development and evolution of the global Internet infrastructure, particularly its core. The perpetual race between the raising demands at improving the speed, scalability, power consumption, cost effectiveness, and hardware lifecycle duration in the Internet core has been driving the engineering response and led to the introduction, adoption, as well as demise of numerous concepts and packet switching technologies over the past two decades.

In the aftermath of extensive research, development, and operational deployment on a historically unparalleled scale, backed by a multi-trillion dollar per year industry, a widely held public perception is that packet switching in the Internet is a solved technical problem, i.e., something which "just works", yet the reality can be different from popular beliefs. For example, on August 12, 2014, due to forwarding tables of certain widely deployed routers being limited to supporting no more than 512K IPv4 prefixes, significant world-wide Internet connectivity outages were observed when the number of prefixes announced in the global routing system exceeded the aforementioned threshold [5] [6]. Approximately a year prior to that event, on the launch date of its latest flagship core router, a dominant network equipment vendor admitted that the product's interface cards based on a newly designed, state-of-the-art application-

specific integrated circuits (ASIC) would not be capable of line rate packet forwarding, but had to be designed oversubscribed at a greater than 1:2 ratio in terms of packets per second forwarding rate capacity [7], far from the full line rate minimum-sized packet forwarding capacity which previously stood among the paramount requirements and performance metrics for core Internet routers since the turn of the century. Even as link speeds progress from 100 to 200 and 400 Gbit/s, the high complexity, risks and enormous costs of developing new ASIC routing hardware are reflected in the difficulty of the industry to deliver improved chips. For example, the Cisco's flagship nPower X1 ASIC from 2013 still has no publicly announced successor at the time of this writing (early 2019).

Software based routers have lost the performance parity with their hardware-based counterparts more than two decades ago. The technological advances in link speeds, in particular 10 Gbit/s Ethernet becoming ubiquitous in recent years, made the already known performance limitations of the traditional network stacks in general purpose operating system (OS) kernels more pronounced. As an example, while a general-purpose OS may successfully forward minimum-sized packets at line rates or emulate networked environments operating in 1 Gbit/s range in real time, it typically struggles with faster link speeds. The measurements the author performed on the FreeBSD operating system (the reference platform used in the experiments throughout this thesis) revealed that its packet forwarding throughput is currently limited to around 1 million packets per second (Mpps) per central processing unit (CPU) core, and that it saturates at even lower speeds if packet filtering is applied, while exhibiting relatively poor scaling properties when running on multicore CPUs. This is consistent with our earlier experiments [8] [9], while others, e.g., Bianco et al. [10]; Bolla et. al. [11]; Brouer [12] observed similar levels of performance in different operating systems. Since a single 10 Gbit/s Ethernet packet flow may require processing up to 14.88 Mpps unidirectionally, the widening performance gap between link speeds and capabilities of contemporary operating systems becomes more palpable.

Advances [13] [14] [15] in improving packet processing efficiency in software have demonstrated that modern commodity CPUs and network interface cards may indeed be capable of absorbing and forwarding packet flows at around 10 Gbit/s line rates. The key to achieving higher throughputs seems to be in blending together several techniques aimed at lowering the effective per-packet handling overhead: processing packets in batches, by carefully engineered data prefetching, minimizing lock contentions, and bypassing the OS network stack altogether. Nevertheless, packet forwarding based on IPv4 (and IPv6) routing lookups is still deemed too demanding a task for a purely software implementation above 10 Gbit/s speeds with large routing tables characteristic for today's Internet exchange points, so the spectrum of different proposals ranges from offloading routing lookups to general-purpose (GP) graphical processing unit (GPU) hardware [14] to distributing the load among multiple physical machines [13] for increased aggregate throughputs.

Examining IP routers [16] as a particular category of packet processors, especially in software based implementations, reveals that IPv4 (and subsequently IPv6) next hop lookups have become a major performance bottleneck already more than two decades ago. The Classless Interdomain Routing (CIDR) [2] principle mandates that an Internet router must select a next hop associated with the most specific network prefix matching the each packet's destination address, i.e., perform a longest prefix matching (LPM) search in the entire routing database. In routers participating in global routing information exchange via the Border Gateway Protocol (BGP) [17] the size of the routing (forwarding) database as of today exceeds 760,000 prefixes, and the prefix count growth shows no signs of abating. As early routing database structures and algorithms (such as [18]) were designed to balance lookup throughput and database updating efficiency, they could not cope with the explosive growth of both BGP table sizes and increasing link speeds. In late 1990s, following a few proposals for more efficient routing lookups in software, such as [19] [20], which soon become impractical due to swift increases in BGP table sizes, both the research community as well as the network equipment industry shifted their focus to hardware-based routing lookup schemes and implementations, a trend which continued up to the present time.

While hardware-based routing lookup methods, for example [21] [22], have solved the performance issues in the past and continue to be applied successfully in various flavors in modern high-performance Internet routers, they generally exhibit several significant shortcomings. First, hardware-based lookup implementations must balance the room for future growth in routing table sizes and link speeds within constrained power (current consumption) and thermal (heat dissipation) envelopes. Moreover, such implementations are generally prohibitively expensive and thus reserved only for carrier-grade Internet routers. And finally, hardware-based routing lookup solutions lack the flexibility which is called for in the emerging virtualization and software defined networking (SDN) [23] scenarios and applications. As a consequence, experimentation and innovation in high-speed routing has gradually become constrained in the realm of only a small and closed circle of network equipment vendors who can afford to develop and build ASIC required for the job. Recent advances in field-programmable gate array (FPGA) integrated circuit (IC) densities and speeds has led to proposals [24] which are moving experimentation with network processing in hardware again closer within the reach of the academic research community and smaller companies, but still at more than an order of magnitude lower speeds as compared to top of the line commercial routers.

This thesis revisits the capabilities and limitations of modern commodity microprocessors for routing lookup applications by proposing a class of efficient longest prefix matching schemes, and subjecting them to a thorough empirical performance evaluation. The practical result is a polyvalent implementation, embodied both as a C library, and as a lookup element in the Click [25] modular router, validated for correctness of operation by comparing the lookups

against the proven (yet slow) PATRICIA trie variant [18] borrowed from the FreeBSD OS, using real-life, full-view BGP snapshots obtained from various open Internet exchange points. The proposed class of schemes outperform other common software IPv4 LPM implementations, particularly when running on multi-core commodity CPUs, where near linear scaling in lookup throughput gains can be observed. The small memory footprint of the proposed lookup structures makes them particularly suitable for both network virtualization scenarios, and for scenarios where cascaded lookups in multiple databases may be required on a single packet forwarding or real-time traffic analysis datapath.

1.2   Thesis overview

The thesis is organized as follows:

The second chapter examines the key properties of contemporary mainstream microprocessors with emphasis on the widening access speed gaps across memory cache hierarchies, and attempts to establish guidelines for data structure and access pattern design aimed at extracting the most of the performance potential from the abundance of execution cores and symmetric multithreading support available in modern CPUs.

The third chapter introduces the Direct-eXtend-Range (DXR) LPM lookup scheme, presents its fundamental principles of operation, motivates the design choices, and discusses implementation tradeoffs. That chapter is based on and contains revised material from the author's initial work [26] as well as from the more recent paper [27].

The fourth chapter discusses further optimization options, which include tradeoffs between memory footprint reduction through data deduplication at the expense of an additional search step, along with implementational microoptimizations which aim at further compensation of memory access latencies in a single instruction stream through batching and prefetching. The impact of the optimizations is dissected through performance analysis under a set of different operating conditions, which also includes a comparison of DXR with other routing lookup schemes.

The fifth chapter discusses the possibilities for DXR's application in data processing datapaths.

Chapter six describes the related work in the field, followed by concluding remarks presented in the seventh chapter.

1.3 Summary

The contributions of this thesis include:

- A new class of algorithms and compact data structures for high-speed IPv4 routing lookups in software;
- Practical implementations of two parametrizable DXR lookup scheme variants, as ready-to-use software libraries or Click elements, and their empirical validation for correctness of operation;
- A thorough analysis of lookup throughput of various DXR configurations running on diverse commodity CPUs;
- Improvements in efficiency of software packet processing datapaths by extracting more performance from spatial and temporal parallel processing capabilities of general-purpose CPUs.

Chapter 2

Packet processing software: a hardware perspective

Internet router vendors had to work hard to start introducing 100 Gbit/s interfaces to the market, yet providing sufficient routing lookup throughput remains among the major challenges in designing router interface cards for even higher speeds (200, 400 Gbit/s and above). To forward minimum-sized IPv4 packets at 400 Gbit/s Ethernet line speed, a core Internet router must perform 579 million routing lookups per second (Mlps) in a database which today consists of approximately 760,000 network prefixes [28] and is continually growing, as shown in Figure 2.1. This requirement is more than four times higher than the capacity of a dedicated, state-of-the-art router ASIC from a dominant vendor, which is reportedly limited to 140 Mlps of unidirectional packet forwarding, or 280 Mlps bidirectional [7].

The evolution of routing ASICs is bounded not only by technological challenges in contemporary silicon design and power dissipation management issues at peak operating conditions, but also by the long, complex and risky development cycles, as well as prohibitively high cost of access to advanced silicon manufacturing processes, which permits only a handful of core router vendors to invest in the increasingly expensive design efforts. Consequently, smaller companies and especially academia are faced with a practically impenetrable barrier of entry to innovation in the field of high-performance router design, which negatively impacts the pace of further technological advances.

Compared to software-based routers, a significant drawback of routing ASICs is their relative inflexibility, which becomes more pronounced as network operators embrace various levels of routing function virtualization, and as the ability to quickly respond to unpredictable malicious threats and security challenges is becoming vital to real-world network operations.

A considerable interest has therefore arisen in (re)exploring the feasibility of utilizing general-purpose CPUs in the forwarding path of high-performance routers, a concept which has been all but abandoned around two decades ago, as it was deemed by far too slow for the rapid increases
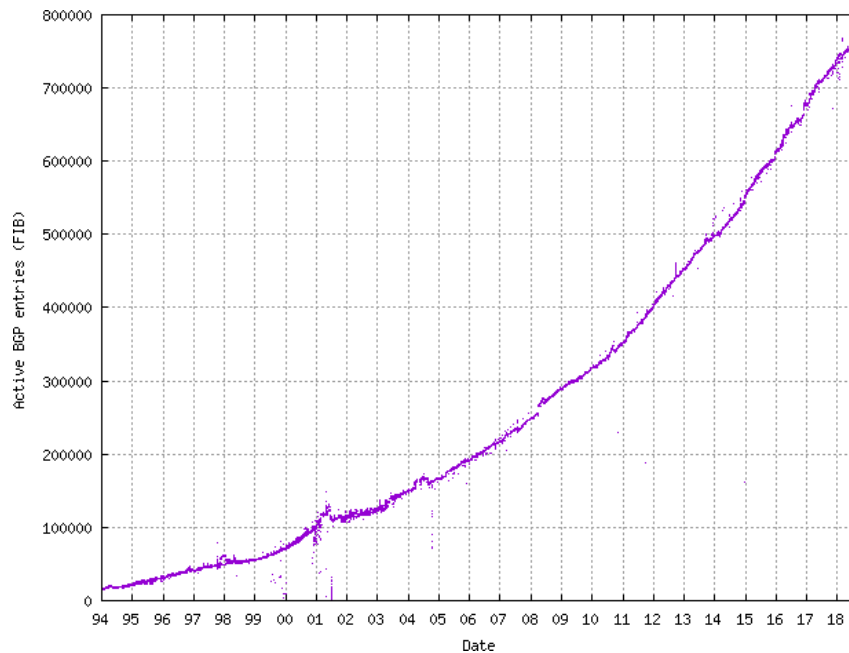
Figure 2.1: The growth of the global IPv4 BGP routing table as of 09/2018. The current trend indicates an annual growth of approximately 50,000 prefixes. Source: BGP Routing Table Analysis Reports [28]

in transmission rates. However, compared to their counterparts from 15 years ago, contemporary multi-core general-purpose CPUs offer a significantly improved performance potential, which an increasing number of recent proposals [29] [30] [31] [26] aim to leverage to offer routing lookup throughputs which rival or exceed the performance of dedicated router ASICs.

The rest of this chapter focuses on key architectural aspects of contemporary commodity CPUs, aimed at identifying both the potential for efficient resolving of queries in large routing databases, as well as identifying pitfalls which should be avoided when designing data structures and the accompanying lookup algorithms.

2.1   Moore's law demise

For the past five decades, the semiconductor technology has been improving at a pace which enabled transistors per area density to roughly double every two years, a rule of thumb which has been colloquially known as the Moore's law [32]. Today's top-of-the-line CPUs are being produced with transistor geometries which are only 14 nm or 12 nm accross. However, as the manufacturers are having a hard time to further shrink the semiconductor process node, there's a widespread consensus that the already faltering Moore's law is about to halt in the near future [33].

As visible in Figure 2.2, even with manufacturing processes considered mature by today's standards, steady increases in microprocessor clock frequencies characteristic for the the past century already hit the ceiling more than a decade ago at around 3.5 GHz [34].
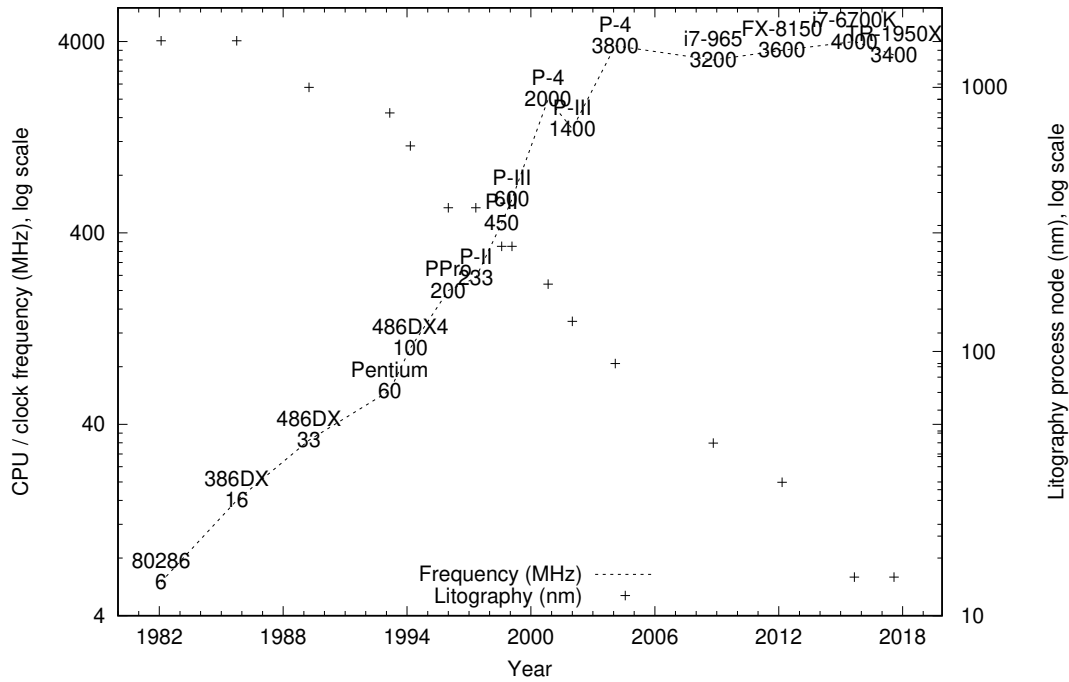
Figure 2.2: Evolution of commodity CPU clock frequencies. Manufacturing process geometry is indicated as crosses vertically aligned to each processor, ranging from 1500 nm in 1982 down to 14 nm node which was first introduced in 2015. Sources: Intel and AMD online product sheets.

Reductions in transistor sizes of more than two orders of magnitude (from 180 nm to 14 nm) over the past decade and a half yielded only a two-fold improvement in processor speeds (from 2 GHz to 4 GHz) due to numerous factors, including wire propagation delays starting to dominate timing budgets at tiny process geometries, ever increasing static leakage currents, enormous dynamic power dissipation per area which became increasingly difficult to manage, timing uncertainties in clock distribution trees, etc. [35].

The same set of technological problems is making advances in ASIC performance just as difficult as with the CPUs. However, the microprocessor industry has so far kept itself ahead of the ASIC world by at least one process technology node, due to the economies of scale which permitted CPU vendors to pour more money in the adoption of newer silicon manufacturing processes, to introduce more advanced products faster to the market, and to compensate for lower yields when moving to finer process geometries.

2.2 Parallelism in contemporary CPUs

Faced with the inability to continue leveraging clock frequency increases for extracting more performance, CPU designers shifted their focus to less rewarding areas and concepts, such as reducing branch penalties by shortening execution pipelines, increasing cache sizes, extracting more single-stream instruction-level parallelism (ILP) by increasing the number of parallel

functional units, introducing advanced memory prefetch and branch prediction units etc.

However, the most notable differentiator between contemporary lower-end, low-power, embeddable CPUs, and their general-purpose counterparts which power today's laptops, workstations and datacenters, is the ability of the later to speculatively execute instructions far beyond of unresolved data dependencies, a technique known as out-of-order instruction scheduling and execution (OoO). Combined with the abundance of memory load / store buffers found in modern CPUs [36], OoO is the key mechanism which offers an effective compensation for a dramatic mismatch between CPU clock frequencies and slow main memory access times, provided that data access patterns are designed, and the programs are written and compiled in a manner which permits speculative execution deep into an instruction stream, i.e., without excessive data interdependencies. Nevertheless, it was established early on that ILP and OoO have firm limits, which may vary widely depending on the workload type and effectiveness of compiler optimizations, but in general for integer applications can be expected to rarely exceed the range between two and three instructions per cycle (IPC) [37].

The recent series of discoveries revealing a whole class of security vulnerabilities in contemporary CPU hardware, which elaborate various side-channel exploits of speculative execution hardware mechanisms [38] [39] [40], prompted responses from CPU vendors which in one form or another disabled or crippled certain speculative execution hardware blocks in attempts to prevent, or at least minimize, the possibility of data leakages to unauthorized applications. It is therefore reasonable to expect that further advances in single-thread computing performance, i.e., in ILP, which have been bound to speculative execution techniques and machinery, will remain marginal if not negative in the foreseeable future.

Realizing that microarchitectural innovations are likely to yield only incremental improvements in effective single-thread performance, approximately a decade and a half ago the whole CPU industry shifted towards integrating multiple processing cores en masse on a single silicon die. Therefore, as shown in Figure 2.3, a modern general-purpose processor consists of several execution cores sharing usually three levels of cache hierarchy, along with external memory controller(s), peripheral interfaces, interrupt routing and multiprocessing synchronization units, among other subsystems.

Each physical execution core typically appears as two virtual cores (threads) to the operating system or application software, although the virtual cores share the same execution units in various time-division schemes, all aimed at finding some useful work to do in times of pipeline stalls due to excessive memory access latencies. Cache blocks closer to execution cores are smaller in size but provide faster response to data access requests. The largest, third level (L3) cache, depicted as a single unit in Figure 2.3, in practice usually comprises multiple smaller blocks bonded together via a high-speed interconnect, which may be of ring, point-to-point, or some other topology, the choice of which varies among CPU vendors and their product
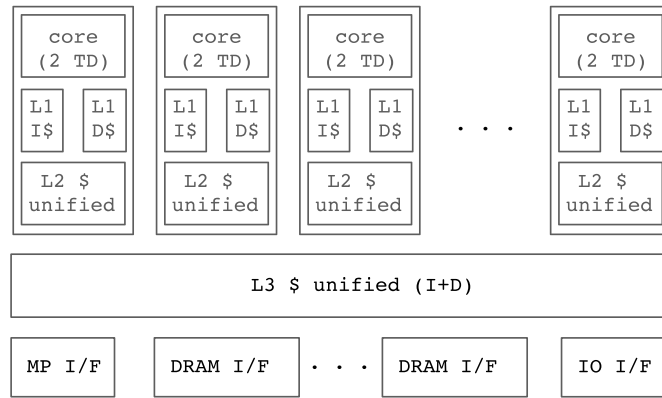
Figure 2.3: A simplified structural diagram of a typical general-purpose contemporary microprocessor, comprising multiple execution cores, a hierarchy of cache memories, and external interfaces. First-level (L1) instruction (I$) and data (D$) caches are separated. Second (L2) and third (L3) caches are unified.

lines [41]. Throughout the memory hierarchy, the main unit of work in hardware interconnects and synchronization machinery is a cache line, an aligned array of 2^N bytes, where the industry appears to have settled on 64 byte blocks as the line size.

Adding more and more cores, regardless whether on a single CPU die or in various multi-die / multi-chip topologies, is far from a trivial endeavor, given that all processing cores in general-purpose computing systems must have the ability to access the whole system's main memory using an uniform addressing scheme, a concept colloquially known as symmetric multiprocessing (SMP). As the number of computing cores rise, the more challenging it becomes to efficiently maintain cache coherency and synchronization throughout the memory hierarchy, which itself gains further complexity by partitioning execution cores among separate silicon dies, each typically with its own (local) memory controller, in topologies known as non-uniform memory access (NUMA) [41].

2.3   Memory hierarchies and latencies

In contrast to increases by several orders of magnitude in microprocessor clock frequencies, as well as both cache sizes and speeds, which took place over the past four decades, contemporary dynamic random access memory (DRAM) has roughly the same access latencies as their ancient predecessors, at around 50 ns. Effective random access latency as observed from a software thread is somewhat higher, as each request and the corresponding data has to pass through all levels of cache hierarchy which further adds up to the total delay [42].

Memory access latencies of several contemporary microprocessors were characterized using a trivial program which populates a large memory pool with random data and then accesses it inside a timing loop in a way which makes each subsequent memory access dependent on the previous one, thus preventing out-of-order execution mechanics from pipelining or inter-

Table 2.1: Characterization of microprocessor cache hierarchies and access latencies. The results are in line with data put forth in Intel's reference manual [36], which estimates L1, L2 and L3 cache latencies at 4, 11 and approx. 34 cycles respectively. The AMD ThreadRipper 1950X CPU includes two silicon dies in a single package, and exhibits increased DRAM latency when accessing physical memory on a die adjacent to the one where the test program was executing, so two DRAM latencies are reported, "local" and "far".

Processor	Year	Cores / Threads	Clock GHz	Level 1 Cache Size KB	Level 1 Cache Latency cycles	Level 1 Cache Latency ns	Level 2 Cache Size KB	Level 2 Cache Latency cycles	Level 2 Cache Latency ns	Level 3 Cache Size KB	Level 3 Cache Latency cycles	Level 3 Cache Latency ns	DRAM Latency cycles	DRAM Latency ns
Intel i5-3210M	2012	2 / 4	2.5	32	7	2.9	256	13	5.3	3072	24	9.7	214	85.8
Intel i3-4150	2014	2 / 4	3.5	32	9	2.6	256	16	4.6	3072	35	10.1	243	69.6
Intel i7-4771	2013	4 / 8	3.5	32	9	2.6	256	16	4.6	8192	38	10.9	260	74.3
Intel i7-5930K	2014	6 / 12	3.5	32	9	2.6	256	15	4.3	15360	51	14.6	263	75.2
Intel E5-2658	2013	10 / 20	2.4	32	9	3.8	256	16	6.7	25600	46	19.2	237	98.8
AMD R7-1700	2017	8 / 16	3.4	32	10	3.0	512	21	6.2	2 * 8192	43	12.7	352	103.8
AMD TR-1950X local	2018	16 / 32	3.4	32	8	2.4	512	14	4.2	4 * 8192	46	13.6	296	87.3
AMD TR-1950X far	2018	16 / 32	3.4	32	8	2.4	512	14	4.2	4 * 8192	46	13.6	452	133.2

leaving multiple memory accesses. The offset from the previous memory address is alternating from positive to negative, which in addition to randomness harvested from previous reads is aimed at preventing the hardware prefetch units from predicting the location(s) of next memory access(es), initiating such reads speculatively, and thereby compensating for a portion of the effective access latency. The results, presented in table 2.1, indicate that on all tested platforms a DRAM read access would cause a software thread to stall for around 250 clock cycles or at least 70 ns, whichever is longer, unless the CPU's execution scheduler would be able to find instructions not depending on the data in flight from the DRAM and execute those instructions out-of-order. The measurements further show that consistently across all observed platforms each cache level has access latencies approximately double of its counterpart closer to the CPU core, and that DRAM access latency is nearly an order of magnitude longer (slower) than a last-level cache hit.

The physical external memory is inherently partitioned into uniform banks (typically 8 internal banks per chip for modern double data rate (DDR) DRAM memory). The banks can be accessed in an interleaved manner, i.e., while one bank is blocked due to access latency, data can be written to or read from another bank on the same memory chip. A memory controller typically uses an interleaved addressing mapping scheme in order to scatter contiguous data blocks over multiple banks, in an attempt to permit multiple threads, or multiple out-of-order requests from a single thread, to make progress while one bank is blocked. Similarly, consumer-grade microprocessors offer multiple (typically two to four) independent physical memory channels, which can be configured for (again) interleaved addressing scheme, permitting further parallelism or better compensation for lengthy random access cycles. The details of low-level memory interleaving machinery have been traditionally opaque to the operating
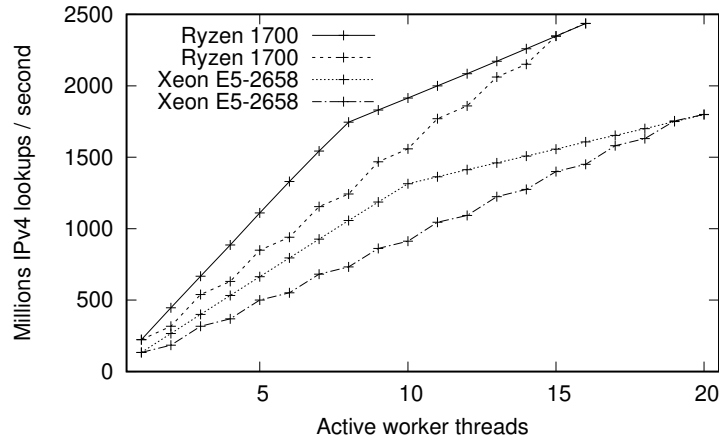
Figure 2.4: The effects of scheduling routing lookup worker threads on virtual SMT cores. In the first experiment, for each of the two microprocessors (AMD Ryzen, Intel Xeon) the workload was scheduled on idle physical cores until all became busy, followed by scheduling a second thread with identical workload on each simultaneous multi-threading (SMT) core, leaving a pronounced "knee" in aggregate throughput increase for the second half of worker threads. In the second experiment, identical worker threads were scheduled sequentially in pairs on idle cores, which can be observed as staircase-like throughput increases.

system or application level programs.

Modern CPU cores include resources for multiple (typically two) SMT hardware threads to share the same execution infrastructure utilizing an opaque time-division scheduling scheme, which permits the core to remain utilized in situations when a single thread can not make further progress while waiting for data to arrive from higher levels of cache hierarchy or from external DRAM [43]. The extent to which such a scheduling scheme can make use of otherwise idle CPU cycles, is both hardware and workload dependent. SMT does not come for free, as it typically mandates partitioning L1 caches in two smaller and independent logical blocks, which unavoidably negatively impacts single-thread performance.

As an illustration, the impact of scheduling threads to SMT virtual cores can be observed in Figure 2.4. In the specific experiments presented here, scheduling a second worker thread on a single physical core yielded an increase in table lookup throughput of around 40%, compared to nearly linear scaling (99.7% increase) when the second thread would be scheduled on a separate and otherwise idle physical core.

2.4 Recommendations

Extracting performance from modern microprocessors calls for careful design of data structures and the accompanying algorithms and access patterns, which have to be closely matched to the underlying physical machine structures. A set of guidelines is summarized here based on aforementioned properties and limitations of contemporary microprocessors, based on observations

obtained from the experiments described in later chapters, as well as backed by reports from other authors in the field.

- Single-thread performance (ILP) on general-purpose computing platforms is stagnating and is unlikely to further improve in the foreseeable future, therefore significant advances in processing performance may be obtained only by designing data structures and algorithms which are suitable for scalable parallel execution on multiple computing cores.

- DRAM random access latency is the major processing performance obstacle in most packet processing applications, which becomes more pronounced with concurrent access from multiple execution cores as congestions form in memory controller blocks [42]. Hence, keeping both shared and thread-local data working sets as small as possible helps to minimize data spilling from higher levels of cache hierarchies to the lower ones, or to the main memory.

- Sharing of read-only, or read-mostly data between execution cores improves L3 cache effectiveness compared to maintaining a separate copy of data for each core (but not necessarily on NUMA topologies). However, multiple threads should avoid sharing the same cache lines with frequent write patterns, as this will trigger excessive synchronization traffic and incur high access latencies until cache synchronization operations complete [44] [45].

- Keeping the data structures small helps not only to achieve high data cache utilization, but permits consolidation and promotion of smaller (4K) memory pages into virtual memory (VM) objects of bigger size (1M to 4M) which are often called superpages, provided the structures may be organized in a contiguous block of memory. Utilization of superpages may significantly reduce the occurrence of translation lookaside buffer (TLB) spills and refills and thus improve performance [46].

- In cases when location of data which will be required within a short timeframe can be computed in advance, the CPU can be instructed to start fetching such data while other unrelated computations take place, thus preempting excessive memory load stalls. The technique should be used with care as data prefetched too far in advance may be displaced from the buffers / cache by the time it is actually needed, effectively doubling the required traffic throughout the cache hierarchy [45] [15].

- Branches hurt performance, especially those dependent on data which has to arrive from main memory. Branch predictors in modern CPUs may be reasonably accurate for a broad range of general-purpose workloads, thus permitting for speculative execution far ahead of unresolved data dependencies. Nevertheless, when dealing with unpredictable data, such as streams of random keys in LPM applications, the value of branch predictors diminishes. Hence, if branches are unavoidable, the data they depend upon should be already in L1 caches.

- With the general-purpose computing industry converging to 64-bit CPU architectures, the lavish use of pointers should be revisited. The problem with pointers is two-fold. First, being 8 bytes wide, pointers carry little useful information for the amount of memory they consume, and often significantly contribute to the large footprint of data structures based on their excessive use. And second, use of pointers promotes scattering of data structures over broad spans of memory addresses, which may exacerbate data access stalls, as random access latencies to main memory are measured in hundreds of wasted CPU cycles. Replacing pointers with (smaller) indices in linear arrays may significantly reduce memory footprints. Today's CPUs provide efficient bit manipulation instructions which make it feasible to instantly extract smaller bit groups (such as array indices) from naturally (power-of-two) aligned fundamental data types, so attempts should be made to compactly encode data, even if that requires unnatural splits inside 16, 32 or 64-bit words.

- Relying on memory access requests to complete within a fixed timeframe is no longer a practical goal on moder CPUs, because memory subsystem latencies cannot be guaranteed to be bounded as the increasing number of cores compete for the shared main memory. Cache synchronization and coherency mechanisms may further interfere with introducing additional delays, and structural competition between SMT contexts for shared execution units on a single CPU core adds more variance to the problem. Last but not least, packet processing systems will typically operate under significant direct memory access (DMA) load originating from network interface card (NIC)s. Therefore, latency variations must be offset by introducing sizeable queues between hardware and software components of packet processing datapaths.

Chapter 3

Direct-Rrange longest prefix matching lookups

The author's interest in improving the performance of software packet datapaths originates from his earlier work on network stack virtualization in the FreeBSD OS kernel [47], and subsequent collaboration on control and data plane integration between the eXtensible Open Router Platform (XORP) [48] and Click [25] platforms.

The routing database (most often colloquially referred to as a "table") in the FreeBSD OS is based on an implementation of Practical Algorithm to Retrieve Information Coded in Alphanumeric (PATRICIA) tree [18], which was originally designed to provide a flexible means of storing, indexing, and retrieving information in a large file, while not requiring rearrangement of text or index as new material is added or deleted [49]. Adoption of such an algorithm for dual-purpose role, i.e., serving as a main routing database, and servicing routing lookups on per-packet basis in the data plane, was an optimal engineering choice at the time when communication links were slow by today's standard. In the era when 10 megabits per second (Mbps) Ethernet and 16 Mbps Token Ring were the fastest (local) network technologies and routing tables were minuscule from today's perspective, supporting packet rates which would rarely exceed 10 thousand packets per second (kpps) was within the reach of an algorithm with search time bounded by the length of the search key, i.e., 32 bits in the particular case of LPM lookups for IPv4 addresses.

However, at the turn of the century this concept was no longer sufficient for sustaining lookup rates in Mpps ranges associated with the ubiquitous 1 gigabits per second (Gbps) Ethernet and the emerging faster link speeds. As the frequencies of routing table updates became many orders of magnitude slower than packet rates in the data plane, a software incarnation of the DIR-24-8 [21] [50] scheme was selected for implementation as a Click IPv4 lookup module. The implementation uses two separate data structure groups, one for database maintenance, specialized for reasonably fast database updating, and the other for performing time-efficient LPM

lookups. The comparatively high computational cost of reconstructing the lookup table from the primary database was compensated by delaying the reconstruction process until the updating of the primary database was completed. For IPv4 address space sections which contain only prefixes with prefix lengths of no more than 24 bits, and those (still) represent the bulk of address space announced in the global routing system, the DIR-24-8 scheme resolves LPM lookups in a single memory access, by using the most significant 24 bits of the key as the index to a precomputed linear array of next hop data. For the small portion of address space corresponding to prefixes with prefix lengths longer than 24 bits, another lookup in an auxiliary table is performed. The scheme was originally designed for dedicated hardware which could be constructed so that multiple DRAM chips could be accessed in a parallel and / or pipelined fashion. However, when implemented in software, the scheme does not take advantage of modern CPUs caches, given that the size of its lookup arrays exceeds cache capacities. Nevertheless, due to its simplicity it still significantly outperforms the traditional radix tries, like the BSD PATRICIA variant, or the conceptually similar trie in the Linux kernel. DIR-24-8 thus remains a popular choice in software packet processing platforms, not only in Click, but also as the standard LPM lookup library in Data Plane Development Kit (DPDK) [45].

It was the experience with implementing the DIR-24-8 scheme as a Click module which made the author aware of the potential of observing the entire IPv4 address space as a continuous sequence of disjoint address ranges, which could be compactly encoded while being arranged in a manner which permits efficient LPM searching. Once it became obvious that the microprocessor industry was heading towards integrating not only a few, but dozens of execution cores onto a single silicon die, surrounded by abundance of reasonably fast and spacious local cache memory blocks, the idea was revisited, prototyped, and gradually further developed over time. The rest of this chapter presents the key ideas behind the proposed LPM algorithm called Direct-Range, shortened as DXR.

3.1 Prefix expansion into address ranges

DXR and its data structures stem from the aforementioned concept of projecting a routing table onto a set of contiguous, non-overlapping address ranges covering the entire address span of a network protocol. The address range containing the search key can then be found through binary search. The idea was already explored before by others, e.g., Lampson and Varghese [20], and according to [51], until 2005. at least a few vendors have implemented this scheme into hardware. In hardware, the use of a wide memory access (to reduce the base of the logarithm) and pipelining (to allow one lookup per memory access) could have made this scheme sufficiently fast for line speeds of that time, i.e., around 10 Gbps. However, the lack of significant follow-up work suggests that this approach was not further pursued, particularly not as a soft-

ware LPM method, possibly due to impractically large size of the proposed data structures (10 to 20 bytes per prefix [52]), combined with relatively high number of memory accesses, and the high number of branches per lookup.

While the concept of binary search on address ranges is not new, the novelty in DXR is in careful encoding of the routing information, so that address range descriptors consume small amount of memory, and are organized in a way which inherently exploits cache organization and hierarchies of modern CPUs in order to achieve high lookup speeds and parallelism of execution on multiple processor cores.

Implementing a universal routing lookup scheme was never among the author's goals: data structures and algorithms described here have been optimized exclusively for the IPv4 protocol. Direct lookups [21], which DXR relies upon to speed up the iterating process before proceeding with binary search on address ranges, are infeasible with IPv6 due to longer keys and sparsely populated address space (0.028%, vs. 66.17% for IPv4) [28].

3.2   Building the search data structure

A network prefix is commonly indicated as a pair {address / prefix length}, where the later is the number of leftmost bits in the address to be matched against the key; the remaining bits are ignored. The LPM principle mandates that among all the prefixes found in a routing database which match the given key, the one with the longest (most specific) prefix length must be used for selecting the target, which is usually a next hop (NH) in routing applications, or some other tag or object relevant for making further decisions on packet's fate.

Consider a sample routing database specified in a canonical {prefix, next hop} notation as shown in Figure 3.1. Building of the search data structure begins by expanding all prefixes from the database into address ranges, and taking into account that more specific prefixes take precedence over less specific ones. This results in a sorted sequence of non-overlapping address ranges which cover the entire IPv4 address space.

Note that the process is irreversible, i.e., unless the information about which table entry corresponds to each range is stored along each range. In practice, if lookup speed is the main design goal, and the size of the lookup structures is inversely proportional to the effectiveness of CPUs caching mechanisms, the cross-reference from ranges to prefix table entries is omitted, as that would consume precious space in the lookup structures. Hence, as it becomes impossible to reverse the transformation using the information remaining in the range table, an auxiliary prefix database as the storage from which the lookup structures can be derived must also be maintained.

Encoding of address ranges in a form more compact than the one shown in Figure 3.1 is possible. Assuming that the ranges are stored in a sorted contiguous linear array, which is

IPv4 prefix	NH		IPv4 address range	NH
1: 0.0.0.0/0	A		0.0.0.0 .. 0.255.255.255	A
2: 1.0.0.0/8	B		1.0.0.0 .. 1.1.255.255	B
3: 1.2.0.0/16	C	=	1.2.0.0 .. 1.2.2.255	C
4: 1.2.3.0/24	D		1.2.3.0 .. 1.2.3.255	D
5: 1.2.4.5/32	C		1.2.4.0 .. 1.2.4.4	C
			1.2.4.5 .. 1.2.4.5	C
			1.2.4.6 .. 1.2.255.255	C
			1.3.0.0 .. 1.255.255.255	B
			2.0.0.0 .. 255.255.255.255	A

Figure 3.1: An example of a transformation of routing information from a canonical prefix table form into a sequence of contiguous address ranges. The process is non-restoring if only next hop information is stored in range descriptors, while references to the originating prefixes are not. The illustrated range encoding requires 10 bytes per range: 4 bytes for range base, 4 bytes for range end, and 2 bytes for a next hop identifier, typically an index in a table of next hop specific objects.

already a prerequisite for efficient binary search, the end address of a range entry can be derived from the start address of the next one. Therefore each entry only needs the **start** address (4 bytes) and the next hop (1 or 2 bytes to index an entry in an external next hop descriptor table), for the total of max. 6 bytes per entry, as illustrated in Figure 3.2.

IPv4 address range	NH		Base	NH
0.0.0.0 .. 0.255.255.255	A		0.0.0.0	A
1.0.0.0 .. 1.1.255.255	B		1.0.0.0	B
1.2.0.0 .. 1.2.2.255	C		1.2.0.0	C
1.2.3.0 .. 1.2.3.255	D	=	1.2.3.0	D
1.2.4.0 .. 1.2.4.4	C		1.2.4.0	C
1.2.4.5 .. 1.2.4.5	C		1.2.4.5	C
1.2.4.6 .. 1.2.255.255	C		1.2.4.6	C
1.3.0.0 .. 1.255.255.255	B		1.3.0.0	B
2.0.0.0 .. 255.255.255.255	A		2.0.0.0	A

Figure 3.2: A more compact encoding of address ranges can be achieved by omitting the upper bound of each range entry, as it can be determined from the lower bound of the subsequent one. Only 6 bytes per address range are required.

Neighboring address ranges that resolve to the same next hop are then merged. In the example shown in Figure 3.3, several ranges resolve to the same next hop "C", so the ranges can be aggregated, thus further saving precious storage space. The process is performed transparently during the rebuild of range table and its computational burden is neglectable. Most importantly, the process is highly effective for real-world routing tables, as often many neighboring prefixes point to the same next hop, due to excessive deaggregation which appears to be a common (although unwelcome and highly discouraged) practice in today's global BGP system [3].

Any routing table containing P prefixes generates no more than $2P+1$ non-overlapping ranges. Provided that such a table is kept sorted, it can be searched in logarithmic time in the number of address range entries. With sizes of global BGP routing tables today exceeding 760,000 prefixes, the worst case results in $1.52 \cdot 10^6$ elements, about 9 Mbytes of memory and
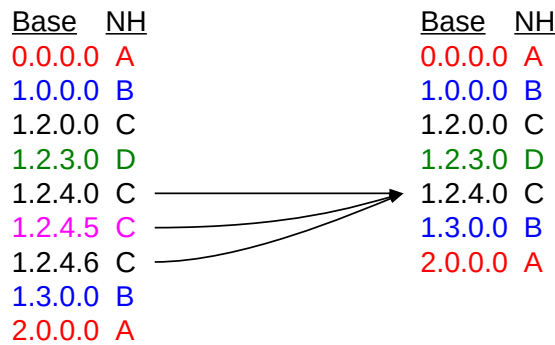
```
Base  NH              Base  NH
0.0.0.0  A            0.0.0.0  A
1.0.0.0  B            1.0.0.0  B
1.2.0.0  C            1.2.0.0  C
1.2.3.0  D            1.2.3.0  D
1.2.4.0  C            1.2.4.0  C
1.2.4.5  C            1.3.0.0  B
1.2.4.6  C            2.0.0.0  A
1.3.0.0  B
2.0.0.0  A
```

Figure 3.3: Neighboring ranges referencing the same next hop can be merged together, reducing the memory footprint of the range table. The process is especially effective due to high degree of prefix deaggregation in today's BGP tables, many of which resolve to the same next hop.

20 search steps, both prohibitively large for the target search times.

3.3 Saving space and time

The next construction step is to shorten the search by splitting the entire range in 2^K chunks, and using the initial K bits of the address to reach, through a direct lookup table, with the range table entries corresponding to separate chunks. The transformation is illustrated in Figure 3.4.

The concept of indexing the lookup tables with a relatively large portion of the IPv4 key was inspired by DIR-24-8 [21], and is effective because of the small key size (32 bit). Compared to DIR-24-8, DXR uses fewer index bits ($K = 16..20$ is used to choose a suitable space/time tradeoff), resulting in smaller lookup tables so that the whole data structure can fit as high as possible in the CPU cache hierarchy.

```
Base  NH           Index  NH
0.0.0.0  A         0000:  A
1.0.0.0  B         00FF:  A
1.2.0.0  C         0100:  B          Base   NH
1.2.3.0  D   K=16  0101:  B          1.2.0.0  C
1.2.4.0  C         0102:  range      1.2.3.0  D
1.3.0.0  B         0103:  B          1.2.4.0  C
2.0.0.0  A         01FF:  B
                   0200:  A
                   FFFF:  A
```
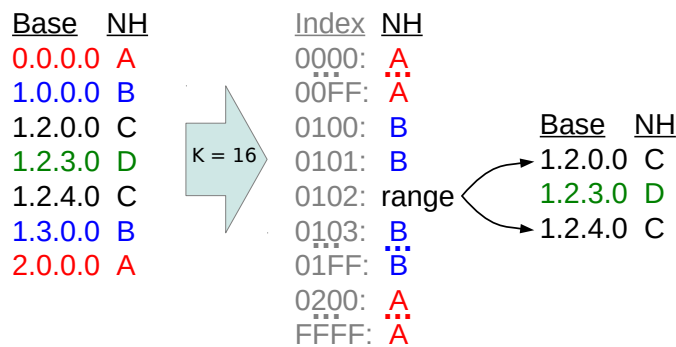
Figure 3.4: Introduction of a direct lookup table reduces the number of iterations required to complete the binary search. In the depicted example a direct lookup table hit is sufficient to resolve search for the entire IPv4 address space, except for 1.2.0.0/16, which has to be resolved via binary search in a corresponding range table chunk. Direct lookup table uses 4 bytes per entry for encoding the range chunk's position in the range table, as well as the chunk's length.

Lookup table entries: Each entry in the direct lookup table must contain the position and size of the corresponding chunk in the range table. 32 bits per entry are used, with 1 bit to indicate the format of the chunk (see below), 12 bits for the size, and 19 bits for the position of the

chunk in the range table. Chunks with only one entry bypass the range table (e.g., the entries from 0x0000 to 0x0101, and entries from 0x0103 to 0xffff in Figure 3.4). A special value for size indicates that the 19 bits are an offset into the next-hop table, i.e., that further iteration over range table is unnecessary. In this case the lookup requires a single L2 or L3 cache access, depending on direct table size.

This arrangement works for up to 2^{19} address ranges after aggregation, and up to 4096 entries per chunk. These numbers should provide ample room for future growth. The decision on how to split the bits can be changed at runtime when rebuilding the tables, and it is trivial to recover extra bits, i.e., by artificially extending chunks so that they have a multiple of 2, 4, 8 entries, with negligible memory overhead.

Range table entries: With K bits already consumed to index the lookup table, the range table entries only need to store the remaining $32 - K$ address bits, and the next hop index. Thus, if $K \geq 16$ bits is chosen for the lookup index, and assuming each next hop can be encoded with 16 bits, 4 bytes suffice for these "long" entries. The example in Figure 3.5 shows the more space-efficient encoding for the chunk covering the range for 1.2.0.0/16.
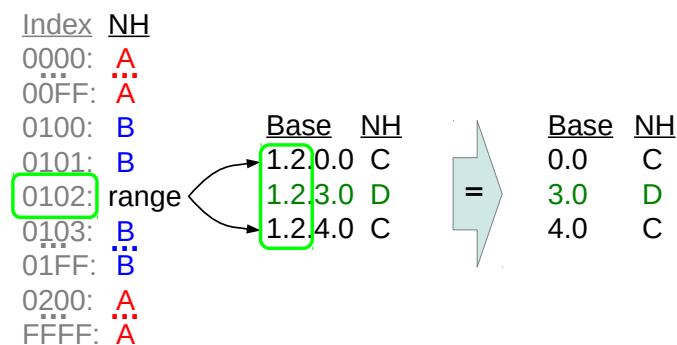
Figure 3.5: Provided that direct lookup table is indexed with at least 16 bits, this permits range descriptors to be further compacted, by allowing range base encoding space to be reduced to only two bytes. By the time the range table is accessed, the leftmost 16 (or more) bits of the search key will already be resolved using the direct lookup table, therefore those bits are entirely redundant information for the rest of the search process. 4 bytes per range descriptor are now sufficient: 2 bytes for range base, plus 2 bytes for next hop info.

A further optimization is especially effective for large BGP views, where the vast majority of prefixes are /24 or less specific, and the number of distinct next hops is typically small, as indicated by prefix length distribution shown in Figure 3.7. If all entries in a chunk contain /24 or less specific ranges, and all the corresponding next hops references can be encoded in 8 bits, a "short" format is used with only 16 bits per entry (the least significant 8 bits of each range base are redundant and do not need to be stored in such cases). Figure 3.6 illustrates this case.

As mentioned, one bit in the lookup table entry is used to indicate whether a chunk is stored in long or short format.
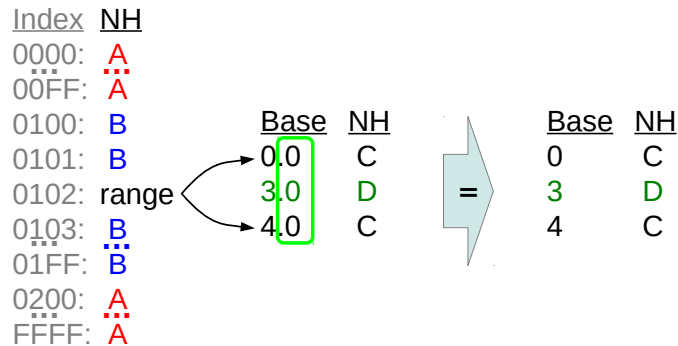
Figure 3.6: Ranges which correspond to prefixes with prefix lengths of 24 bit or less are inherently guaranteed to have all range bases with the 8 least significant bits set to zero. If all ranges in a chunk also include references to next hops which can be encoded no more than 8 bits, the range descriptor's memory footprint can be reduced from 4 to only 2 bytes.
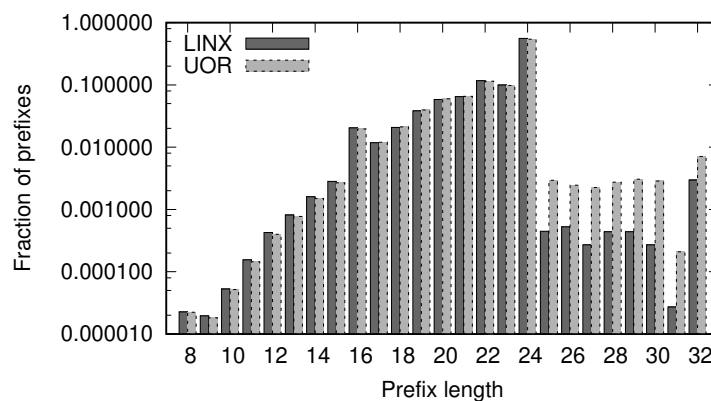

Figure 3.7: Distribution of prefix lengths for two of the linx.routeviews.org [53] BGP snapshots, April 2017. The majority of prefixes have a prefix length of 24 bits or less. The University of Oregon snapshot has an unusually high proportion of prefixes with prefix lengths higher than 24, compared to other BGP snapshots.

3.4   Lookup algorithm

As described in the previous section, the two DXR's main lookup structures are the fixed-size (direct) lookup table, accompanied by the variable-length range descriptor table. Figure 3.8 shows the arrangement for $K = 16$, followed by C definitions of each table's elements.
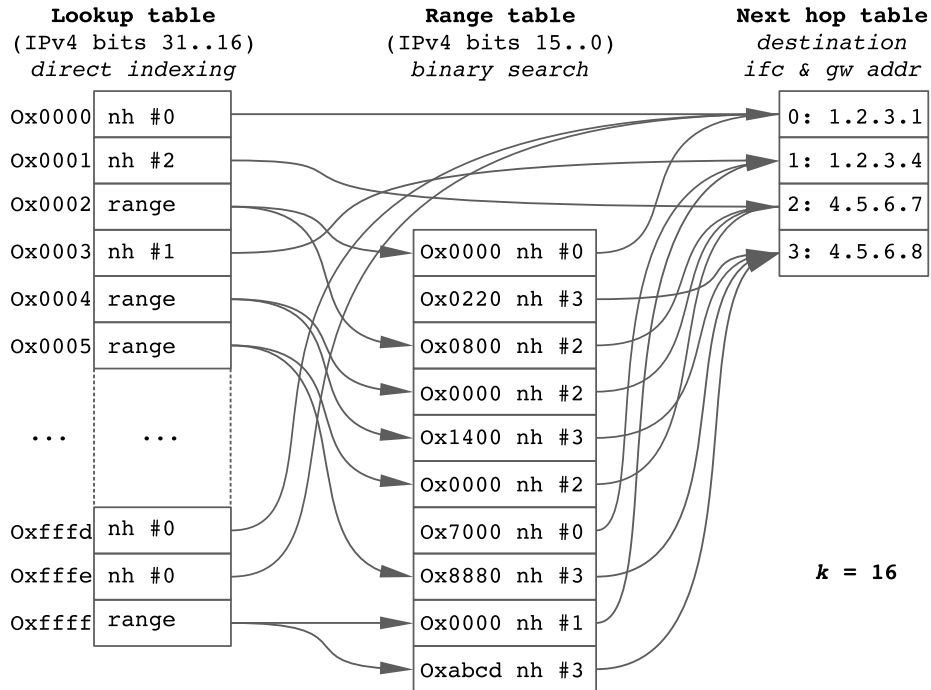
Figure 3.8: An example of DXR's data structures. The lookup table has fixed size (2^K 32-bit entries), whereas the size of the range table is variable.

```
#define DESC_BASE_BITS 19

struct direct_entry {
        uint32_t
                fragments:(31 - DESC_BASE_BITS),
                long_format:1,
                base:DESC_BASE_BITS;
};


struct range_entry_long {
        uint16_t nexthop;
        uint16_t start;
};


struct range_entry_short {
        uint8_t nexthop;
        uint8_t start;
};
```

The lookup algorithm is trivial: the *K* leftmost bits of the key are used as an index in the lookup table. If the entry points to the next hop, the lookup is complete. Otherwise, the (position, size) information in the entry select the portion of the range table on which to perform a binary search of the (remaining bits of the) key.

Since range table entries are small (2 or 4 bytes), as the binary search proceeds, it becomes more probable that the entries which remain to be accessed have already migrated from L3 / L2 to the L1 CPU cache, which inherently further speeds up the lookup process. Given that cache line size is today 64 bytes long on virtually all general-purpose microprocessor platforms, the small range entry size (2 or 4 bytes), means that the last 5 (or 4) search iterations are guaranteed to be serviced from L1 cache, assuming the most compact DXR configuration (D16R) and today's distribution of prefixes and prefix lengths in the global BGP databases.

A similar effect will speed up fetching of descriptors for biger ranges from the DRAM. All DRAM memories are phyisically organized in pages, which in modern silicon range from 512 bytes to 2 Kbytes. The first access to a DRAM page incurs a significant latency overhead, due to the (slow) time required to fetch the entire page into a matchingly wide SRAM register. Once the page data is in the register, it may be transferred from the DRAM chip towards the CPU with significantly lower delay (around 20 ns) compared to the first access initiating "opening" of a page, which typically lasts for 40 to 50 ns. The extent to which the algorithm will be able to exploit hitting the already open DRAM pages depends on how the memory controller maps multiple DRAM chips into the physical address space visible to the CPU, as different implementations will follow different memory block interleaving strategies.

Given the lookup algorithm's simplicity, and the brevity of its C implementation, it is presented here in its entirety. The key part, the binary search in a range chunk, tracks upper and lower bounds of already probed range space, and narrows the pool of remaining ranges until the lower and upper bounds converge. The body of the search loop is abstracted as a macro (shown below) so that it can be applied for iterating over both short and long format ranges without code duplication. Moreover, this approach allows for experimenting with manual loop unrolling which may slightly improve the performance on some CPU microarchitectures. The macro was chosen instead of an inline function because the former gets more efficiently blended into loops by the current generation of C / C++ compilers (gcc, clang).

```
#define DXR_LOOKUP_STAGE                                        \
        if (masked_dst < range[middle].start) {                 \
                upperbound = middle;                            \
                middle = (middle + lowerbound) / 2;             \
        } else if (masked_dst < range[middle + 1].start) {      \
                lowerbound = middle;                            \
                break;                                          \
        } else {                                                \
                lowerbound = middle + 1;                        \
                middle = (upperbound + middle + 1) / 2;         \
        }                                                       \
        if (upperbound == lowerbound)                           \
                break;
```

In the actual lookup function implementation shown below, individual bit fields are extracted / masked manually from packed structures, in order to circumvent compiler inneficiencies, which in some cases can be observed as redundant bitmask or shift instructions being emited.

The base index in the range table (rt) is extracted from the direct table (dt) lookup. If the extracted index amounts to the reserved value ($BASE\_MAX$), the next hop information is encoded in the remaining bits of the (dt) entry, and the lookup completes.

Otherwise, depending on the format of the range chunk, different computation is performed for determining the starting values for upperbound, lowerbound and middle for short and long format ranges, as the former is twice more compact than the later. The iterative search then begins, which is expanded from the *DXR_LOOKUP_STAGE* macro within (seemingly infinite) do-while loops, which are terminated from within the macro as soon as the binary search converges.

```
int dxr_lookup(uint32_t dst, struct direct_entry *dt,
    struct range_entry_long *rt)
{
        uint32_t *fdescp; /* range fragment descriptor pointer */
        int32_t nh;
        uint32_t masked_dst, uint32_t upperbound, middle, lowerbound;

        masked_dst = dst & DXR_RANGE_MASK;
        fdescp = (uint32_t *) &dt[dst >> DXR_RANGE_SHIFT];

        lowerbound = *fdescp;
        nh = lowerbound >> (32 - DESC_BASE_BITS); /* nh == .base */
        if (nh != BASE_MAX) {
                if (lowerbound & 0x1000) { /* .long_format set? */
                        struct range_entry_long *range;

                        upperbound = lowerbound & 0xfff; /* .frags */
                        range = &rt[nh]; /* nh == .base */
                        middle = upperbound / 2;
                        lowerbound = 0;

                        do {
                                DXR_LOOKUP_STAGE
                        } while (1);
                        nh = range[lowerbound].nexthop;
                } else {
                        struct range_entry_short *range;

                        middle = lowerbound & 0xfff; /* .frags */
                        masked_dst >>= 8;
                        range = (struct range_entry_short *) &rt[nh];
                        upperbound = middle * 2 + 1;
                        lowerbound = 0;

                        do {
                                DXR_LOOKUP_STAGE
                        } while (1);
                        nh = range[lowerbound].nexthop;
                }
        } else
                /* nexthop is encoded in the fragments field */
                nh = lowerbound & 0xfff; /* .frags */

        return (nh);
}
```

3.5   Updating

DXR's lookup structures store only the information necessary for resolving LPM searches, so a separate database which stores detailed information on all the prefixes is required for rebuilding the lookup structures. DXR uses the proven PATRICIA radix tree implementation already available in FreeBSD as a reasonably portable C library (usable in both kernel and user space applications) which is well suited for that purpose, although in principle other routing database formats could work with DXR as well.

Updates (additions and removals of individual prefixes) are handled as follows: first, updates covering multiple chunks (prefix length $< K$) are expanded in smaller entries, each covering a single chunk; then each chunk is processed independently and rebuilt from scratch.

The process of rebuilding a chunk begins by finding the best matching route for the first IPv4 address belonging to the chunk, translating it to an range table entry, and storing it on a heap. The algorithm then continues to search the primary database for the next longest-matching prefix, until the search falls out of the scope of the current chunk. As more prefixes are found, if they point to the same next hop as the descriptor currently on the top of the address range heap, they are simply skipped over, until a prefix pointing to a different next hop is encountered. This allows for very simple yet surprisingly efficient aggregation of routing information, and is the key factor which contributes to the small footprint of the lookup structures. If the range table heap contains only a single element when the process ends, the next hop can be directly encoded in the lookup table, and the range table heap may be released.

The rebuild time can be reduced by coalescing multiple updates into a single rebuild (this was implemented by delaying the reconstruction for several milliseconds after the first update is received), and processing chunks in parallel, if multiple cores are available (this does not reduce the total work but does reduce the wall clock time).

A suitable tradeoff between lookup table size and reduction in number of remaining iterative search steps can be tuned by choosing an appropriate value for K. As reducing the effective memory access latency depends on enabling data structures to reside as high as possible in the CPU cache hierarchy, in practice the most useful choices for K have been shown to be in 16 to 20 range, which corresponds to lookup structure footprints from around 1 to 5 MBytes, or 1.76 to 7.32 bytes per prefix, as shown in Table 3.2.

Table 3.1 shows the fraction of IPv4 address space which requires n binary search iterations in the range table, for a range of DXR configurations depending on parameter K. As K increases, both the size of the range table, and the fraction of address space which has to be resolved via binary search gets reduced.

For the most compact configuration, D16R ($K = 16$), in the worst (and highly unlikely) case resolving a lookup may require 8 iterations. Assuming that the corresponding range chunk

Table 3.1: Fraction of addresses that require exactly n iterations in the binary search ($n = 0$ means a match in the direct lookup table). Size of the range table is shown in colum 2. Distribution based on the September 2018 snapshot of a forwarding information base (FIB) from a Equinix Internet Exchange (EQIX) router.

K	bytes	0	1	2	3	4	5	6	7	8
16	809308	71.413%	2.090%	5.127%	5.177%	5.165%	5.075%	3.911%	1.973%	0.069%
17	804060	77.556%	2.145%	5.051%	4.724%	4.449%	3.721%	2.229%	0.126%	0%
18	784448	83.136%	2.269%	4.556%	3.894%	3.511%	2.399%	0.234%	<0.001%	0%
19	735420	88.229%	2.007%	3.594%	3.290%	2.466%	0.412%	0.001%	0%	0%
20	625428	92.140%	1.555%	3.210%	2.452%	0.639%	0.002%	<0.001%	0%	0%
21	372600	94.863%	1.800%	2.412%	0.921%	0.003%	<0.001%	0%	0%	0%
22	81912	97.299%	1.370%	1.325%	0.005%	0.001%	<0.001%	0%	0%	0%
23	10012	98.697%	1.293%	0.008%	0.001%	<0.001%	0%	0%	0%	0%
24	4548	99.996%	0.003%	<0.001%	<0.001%	0%	0%	0%	0%	0%

would be using the long format encoding (4 bytes per range), and given that cache line size is 64 bytes long, the data for the last 3 search iterations will be serviced from L1 cache. In other words, worst-case number of L3 cache memory references will include a direct lookup table reference, plus additional 5 accesses associated with the binary search, for a total of 6 of L3 and 3 L1 cache accesses in the final stage of the search process. Given the estimated L3 latency of 13.6 ns and L1 latency of 2.4 ns (Table 2.1), the total memory access latency of such worst-case scenario amounts to 88.8 ns.

For the configuration which yielded the best average throughput on most CPUs (see next section), D21R ($K = 21$), worst-case iteration count is reduced by 3 steps (with extremely low occurence probability), i.e., by 3 L3 and 3 L1 accesses, which translates to 48 ns. A naive analysis would translate this latency to approximately 20 Mlps of worst-case throughput, but in practice, if a traffic pattern would focus on the tiny address space portion which requires the most extensive binary search, the corresponding range chunks would migrate towards the L1 cache, reducing the latency to $6 \cdot 2.4$ or 14.4 ns, which corresponds to approximately 69.4 Mlps. Given that OoO machinery might further hide some of that latency by pipelining consecutive lookup requests, but that more time will be lost in (mispredicted) branches throughout the iterative search process, worst-case lookup latency averaging in range of 20 ns could be taken as a reasonable estimate.

The original DXR code was written in ANSI-C and targeted for execution inside the FreeBSD kernel. The algorithm was recoded in C++ and implemented as a processing module inside the Click modular router [25]. For simplicity, it was decided to retain the original BSD radix tree code [18] as a backing store for the routing table, hence the ANSI-C implementation of BSD tree was encapsulated in an additional Click class / element. Such an approach simplified the

Table 3.2: Characterization of the DXR data structures for several different IPv4 routing tables. The time to rebuild the lookup structures from scratch is also given, in milliseconds.

| Table | IPv4 | Next | k = 16 (D16R scheme) | | | | k = 20 (D20R scheme) | | | |
| | | | Footprint | Direct | Range fragments | | Build | Footprint | Direct | Range fragments | | Build |
snapshot	prefixes	hops	(bytes)	coverage	short	long	(ms)	(bytes)	coverage	short	long	(ms)
PAIX 2014	504818	58	879964	75.6 %	269536	19687	70.1	4636992	94.0 %	210208	5568	316.0
EQIX 2014	493049	58	807280	77.6 %	236388	18090	92.0	4563208	94.7 %	177834	3309	687.2
LINX 2014	513644	239	954568	75.3 %	269850	38181	98.7	4706452	93.8 %	237728	9173	758.9
PAIX 2017	675791	85	970664	73.3 %	313376	20442	94.1	4694696	93.0 %	237316	6440	501.0
EQIX 2017	672790	159	952136	73.7 %	302256	21370	120.0	4663640	93.2 %	220446	7111	911.3
LINX 2017	663729	560	1170504	73.0 %	280318	86931	120.4	4856504	92.7 %	293006	19047	900.4
UOR 2017	713253	34	1192072	72.4 %	287726	88619	102.1	4873896	92.6 %	286844	26476	524.3
EQIX 2018	739561	148	1071452	71.4 %	377030	13812	95.0	4819732	92.1 %	305498	3608	757.0

construction of a portable synthetic testbench. Implementation inside Click also resulted in instant portability to other operating systems, such as Linux. The new implementation allows multiple independent DXR instances to coexist inside a single Click configuration, which will permit future experimentation focused on network function virtualization implications.
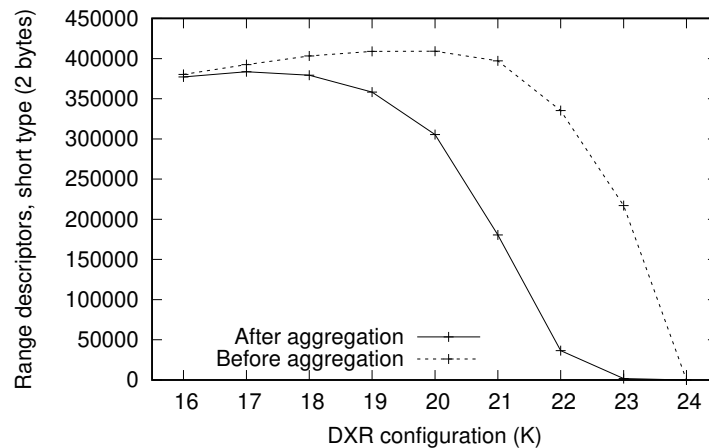
Figure 3.9: Aggregation of short type range descriptors

As an improvement over the original implementation, the revised DXR/Click version implements chunks as reference counted objects, which reduces the lookup structure's memory footprint by the size of each identical chunk copy. As the chunks are being rebuilt, new chunks are compared against the existing ones, and if a match is found, a reference to an existing chunk is used, while the newly allocated range entries are discarded. In practice, this has been shown to yield virtually no impact with most compact ($K = 16$ or $K = 17$) lookup structure configurations. However, for higher values of K the size reduction of range table becomes measurable, especially for the chunks with short type encoding. The effect becomes significant beyond $K = 19$, and compression ratios of nearly 90% are achieved for $K = 22$, as shown in Figure 3.9.

The compression effectivenes improves with higher values of *K* because the number of ranges per chunk decreases, and as the chunks get smaller, more of them happen to have an identical layout for the lower bits of the lookup key.
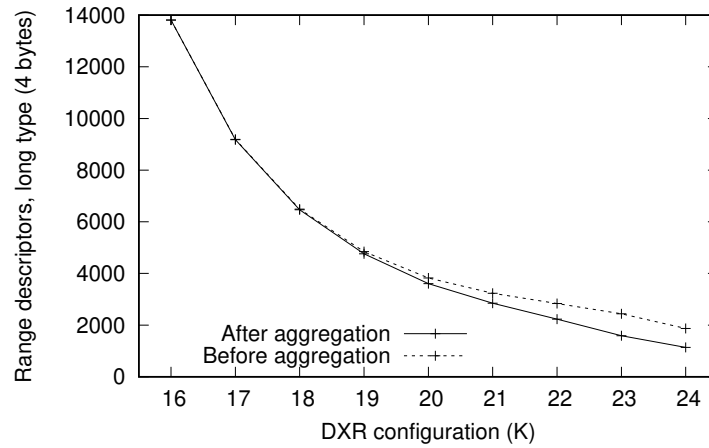
Figure 3.10: Aggregation of long type range descriptors

The process is by far less effective for the chunks of long type, as shown in Figure 3.10. Given that short type ranges dominate in volume over long type ones by an order of magnitude (at least until $K = 23$), the ineffectivenes of long type range aggregation is not detrimental for the overall data deduplication success.

The overall memory footprint of lookup structures for a range of DXR configurations is shown in Figure 3.11.
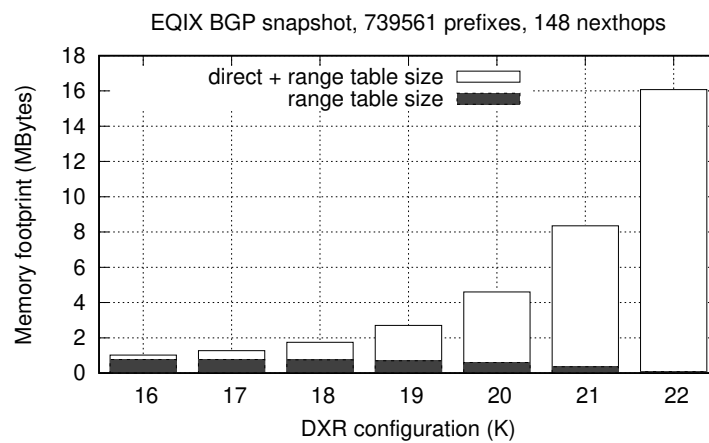
Figure 3.11: DXR memory footprint as a function of K from 16 to 22, which yield lookup structures of practical sizes. As K is increased, the size of range chunks decreases, and so does the number of iterative search steps required for completing the lookup, at the expense of direct lookup table's exponential growth. Configurations with K > 22 are not shown, as their size, which exceed the capacity of even largest L3 caches of contemporary CPUs, makes them useless for real-world applications.

Finally, the DirectIPLookup Click element which embodies the DIR-24-8 lookup scheme [21] was also reimplemented from scratch. This was necessary since the original DirectIPLookup

implementation (dating from 2005) could not build lookup structures corresponding to contemporary databases of around 760,000 prefixes in reasonably short timeframes. A fully functional DirectIPLookup element permitted us not only to compare DXR against DIR-24-8 performance-wise, but to check them both for correctness against the proven BSD radix tree implementation, which led to discovery of several subtle bugs in the original DXR version, which were subsequently rectified.

3.6   Performance evaluation

Request patterns: Rather than relying on a specific traffic traces which typically exhibit address locality, lookup performance was measured using synthetic streams of random IPv4 addresses. Large arrays (500 million entries) of precomputed random IPv4 keys were used, uniformly distributed across the entire address space excluding multicast and other reserved ranges. The precomputation removed the cost of random number generation from the measurement. During the tests, each CPU core issued lookup requests in a tight loop, reading keys from an independent section of the array in order to avoid multiple worker threads synchronizing on the same stream of keys, which could artificially increase CPU cache locality.

Lookup results were stored in a separate array, otherwise the CPU's OoO machinery could discard the results once the destination register would become overwritten, which would yield unrealistically high throughput results. Hence, all the reported throughput levels include the overhead of fetching the inbound keys, as well as storing the results back to memory.

Three different request patterns were used:

- **RND** (random): each key is looked up only once. The test loop has no data dependencies between iterations, so that CPUs with OoO capabilites can achieve increased single-thread throughput, primarily by pipelining memory access requests, which is enabled by processing of lookup requests in batches. This workload tries to replicate the worst case for a high performance router working in the core of the network, where flows are spread across the entire IPv4 address space with minimum or no temporal locality.

- **SEQ** (sequential): same as RND, but the timing loop has an artificial data dependency between iterations so that requests become effectively serialized. This pattern is meant to emulate the behavior with individual lookups, or cases when a decision / branch must be taken immediately based on the lookup result. Such behavior is also inherent to CPUs without efficient out-of-order execution capabilities, where no instructions may make progress ahead of unresolved memory fetches.

- **REP** (repeated): each key is looked up 8 times interleaved with 7 other keys from a sliding window progressing over the precomputed array of random keys, constructed in the same

way as for the RND test. This represents an optimistic situation where the majority of requests can be served from L1 cache, but the sliding window method attempts to defeat the CPU's branch predictor from becoming overly trained and thus unrealistically precise. This test attempts to mimic what might happen on a router handling a small number of interleaved flows. The throughput levels observed using the REP test are in most cases higher compared to RND tests using the same FIB database and DXR configuration. In author's opinion those results are of questionable value, since it is the RND test which subjects the CPU caches and memory subsystem to the highest stress levels. However, since a trend in recent publications is observable where other authors tend to rely on traffic "patterns" and "traces" for relaxing the pressure on CPU caches in their experiments, and reviewers apparently have no objections to such twists, benchmarks with certain locality in the key stream may be useful for comparision against proposals and reports which do not provide any insight on throughput with random lookup keys.

Databases: Prefixes with lengths of /24 dominate in all snapshots, followed by less specifics, similar to two snapshots shown earlier in Figure 3.7. Since networks with prefix lengths more specific than /24 typically originate from peering links between BGP speakers in Internet exchange points, they are less often globally announced. A 2017 snapshot from the University of Oregon (UOREG) stands out from the rest by including a disproportionate amount of prefixes with prefix lengths higher than 24, which also contributes to the total number of prefixes which in that particular snapshot exceeds the average of other Internet exchange points by around 40,000, an anomaly which we did not further investigate. Instead, a London Internet Exchange (LINX) snapshot from 2017 was chosen as the baseline for most of the experiments, since it includes the largest number of unique next hops, and as such minimizes the opportunities for route / range aggregation. More recently performed tests are based on snapshots from September 2018. Nevertheless, an experiment where the impact of routing table properties (the number of prefixes and next hops) on aggregate lookup throughput was explored shows that the performance variations with different snapshots are minimal (around 5%), as visible in Figure 3.12.

Hardware / OS: Most of the experiments and measurements were conducted on an 8-core, 16-thread, 3.4 GHz AMD Ryzen 7 1700 with 8 Gbytes of RAM, and an quad-core, 8-thread, 3.5 GHz Intel i7-4771 with 16 Gbytes of RAM. Several other machines were also used for certain tests, depending on their availability. Cache hierarchy characterization and other key info about test machines were presented earlier in Table 2.1

Each timing test ran for 10 seconds (which in many cases amounts to a billion lookups per CPU core, or more), averaging the throughput over the entire test. While the measurements had very low variance (less than 1%), at these speeds even small changes in the test code, compiler optimizations, or memory configuration may lose or gain 10% to 20% in the performance. For

this reason, all tests ran on the same platform/compiler (FreeBSD 11.1-RELEASE, amd64, clang/LLVM compiler version 6.0.0).
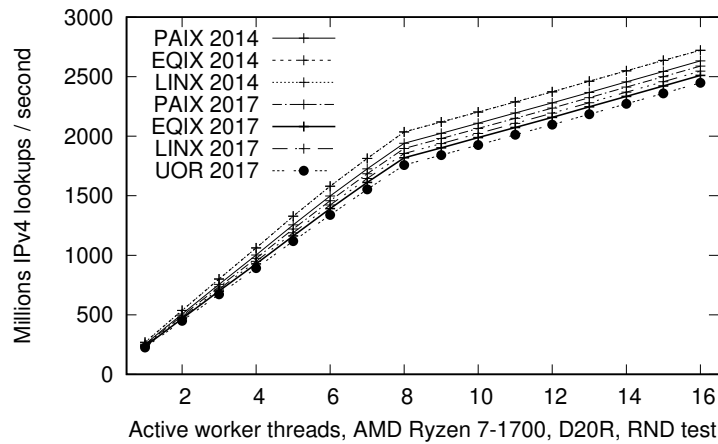

Figure 3.12: Aggregate lookup throughput for different BGP table snapshots as a function of the number of active worker threads subjected to streams of uniformly random lookup keys. The reduced slope in throughput increase beyond the 8th worker thread is due to a lower contribution of thread pairs scheduled on simultaneous multi-threading (SMT) virtual cores.
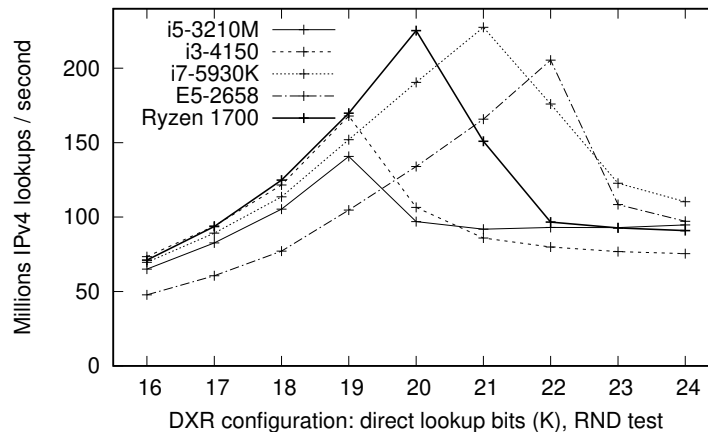

Figure 3.13: Single-thread performance for different DXR configurations. LINX 2017 snapshot, RND test (a stream of uniformly random keys). CPUs with more spacious L3 cache exhibit peak performance with more bits of the lookup key used for direct indexing, as long as the overall size of the lookup structures is lower than L3 cache size. Depending on the CPU, the performance sweetspot is the DXR configuration for which the structures expand to approximately 0.5 of the CPU's L3 cache size.

Figure 3.13 shows how the arrangement of DXR structures (parameter K) influences peak single-thread lookup throughput, driven by a stream of uniformly random (RND) keys. The same experiment was repeated on five machines with different core counts, cache sizes, clock speeds and memory access latencies, as shown in Table 2.1. The machines with bigger L3 caches benefited more from DXR configurations with higher values of K, but as soon as the size of lookup structures approached or exceeded the cache size, lookup throughputs collapsed due to excessive latencies of fetching data from the external DRAM. As there were no dependencies

between successive queries, even when data has to be fetched from cache layers far from the processor core or even DRAM, out-of-order execution mechanics could begin to resolve the next key, thus effectively interleaving several lookups.
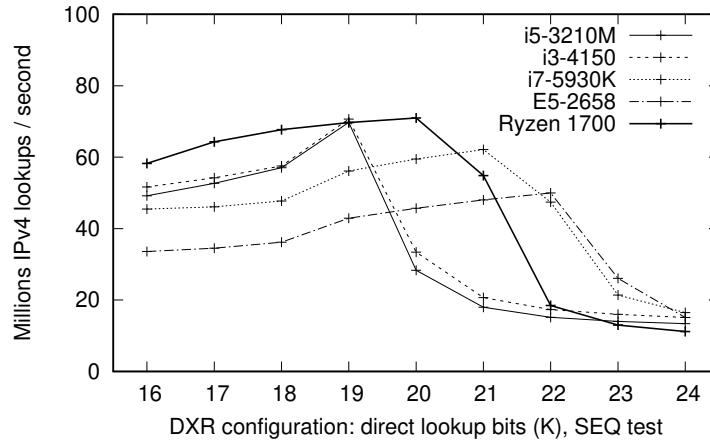
Figure 3.14: Single-thread performance for different DXR configurations. LINX 2017 snapshot, SEQ test (each key is subjected to the lookup process 8 times in a sliding window of random keys).

Figure 3.14 shows the effects of introducing artificial dependencies between successive lookups by logically XORing each key with the result of the previous query. In such a setting the CPU's out-of-order scheduler is prevented from pipelining memory reads since the address of each memory read could not be computed before the previous lookup was completely resolved. The top effective throughput was significantly lower compared to operation on independent keys (70 Mlps vs. 235 Mlps).
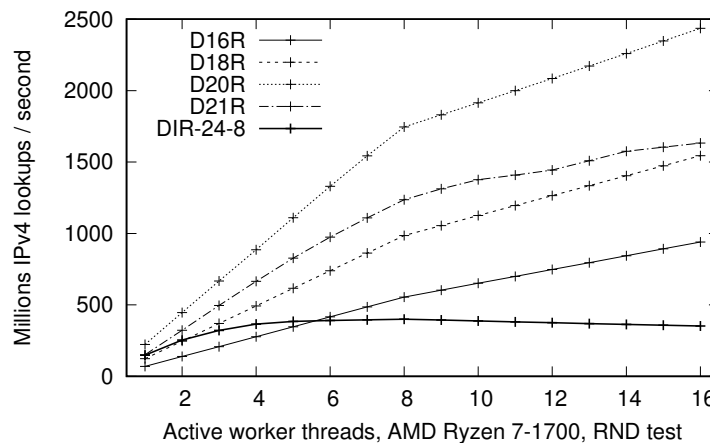
Figure 3.15: Aggregate lookup throughput for DIR-24-8 and four different DXR configurations as a function of the number of active worker threads, using LINX-2017 table snapshot (663729 prefixes, 560 nexthops) and streams of uniformly random lookup keys. The throughput with DXR configurations up to $K = 20$ scales nearly linearly with additional CPU cores due to lookup structures fitting in cache hierarchies, while D21R and DIR-24-8 are limited by DRAM's random access throughput.

The choice of parameter K determines how the algorithm scales on multiple execution cores, as shown in Figure 3.15. The graph shows the increases in lookup throughput with additional
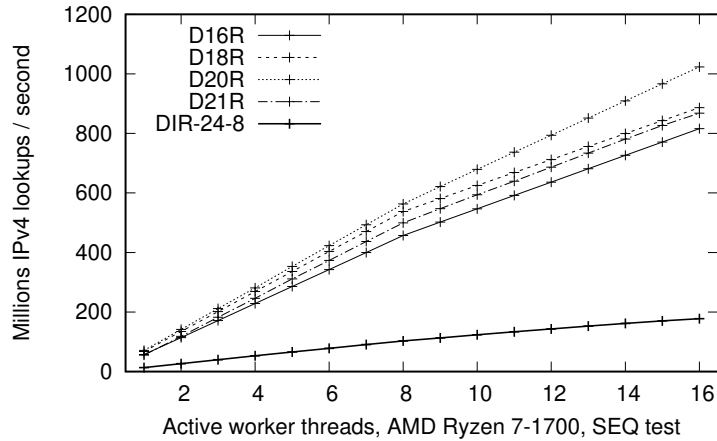
Figure 3.16: Aggregate lookup throughput, LINX-2017 table snapshot, uniformly random lookup keys with artificial dependencies. Working threads operate independently, but within a thread each lookup must be resolved before proceeding to the next key.
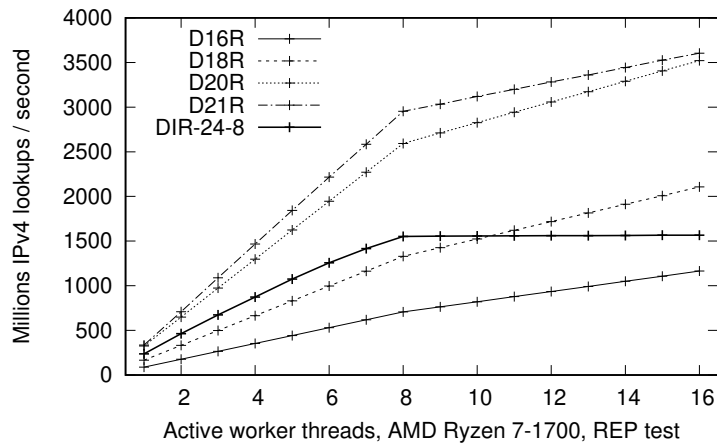
Figure 3.17: Aggregate lookup throughput, LINX-2017 table snapshot, uniformly random lookup keys with repeated queries. Each key within a sliding window is looked up 8 times, in an attempt to emulate locality in traffic patterns, while still preventing CPU's branch predictors to become over-trained and yield overly optimistic results.

worker threads on an AMD Ryzen 7-1700 machine, using a stream of independent, uniformly random lookup keys (RND test) as a stimulus. Similar trending was observed on other machines as well, with different choices of K yielding the best overall throughput which can be correlated to the size of CPU caches, as previously shown in Figures 3.13 and 3.14. Only the graph for the AMD machine is shown, since it was the most modern machine available for experimentation, and since it yielded the top aggregate throughput among all tested CPUs at 2,449 Mlps, i.e., 2.45 <u>billion</u> lookups per second. Another important result that can be observed in Figure 3.15 is how well the DXR scheme scales compared to DIR-24-8, which has a working-set footprint of around 33 MB, and which thus does not fit the lookup structures in CPU's caches. While DXR and DIR-24-8 yielded comparable throughputs on a single CPU core, as soon as more worker threads were introduced, the throughput of DIR-24-8 saturated and even slightly collapsed due

to the inability of the DRAM subsystem to service random access read patterns beyond a certain threshold.

Again, introducing artificial dependencies between subsequent queries had negative impact with multiple worker threads, as shown in Figure 3.16 (SEQ test). Conversely, Figure 3.17 illustrates how the tested algorithms could behave when subjected to traffic patterns with certain degree of locality (REP test), which is a natural property of regular (non-malicious) transfers in the Internet. In this test a small sliding window was introduced under which random keys were repeatedly used, which permitted the lookup structures to be reused for several times after they migrated to L1 cache, before being displaced by other random keys. Both DXR and DIR-24-8 show an increase in overall throughput under such conditions, though DIR-24-8 benefits more from traffic locality, as its pressure on the DRAM subsystem gets significantly reduced.

Further tests were aimed at determining the levels of pressure on the DRAM subsystem by tracking last-level-cache miss counters [54]. The experiments were conducted on several Intel machines which have mature support for accessing hardware performance monitoring counters (HWPMC). Unfortunately, the drivers for HWPMC tracking on AMD CPUs turned out to produce inconclusive and unreliable results, which where therefore not further pursued.
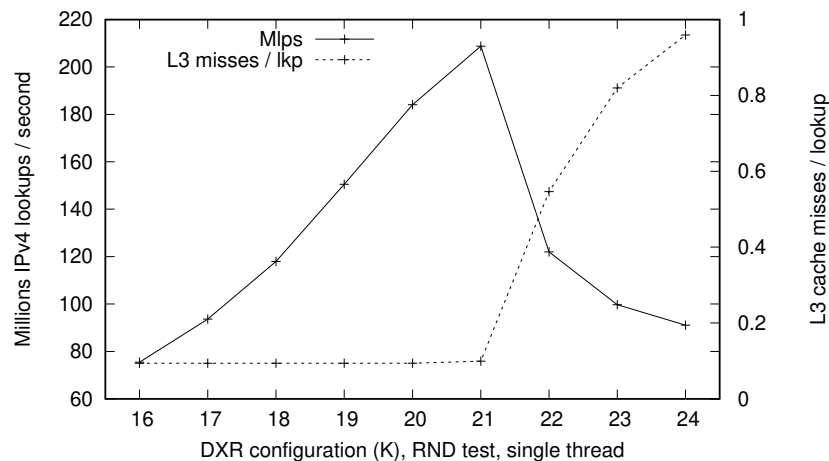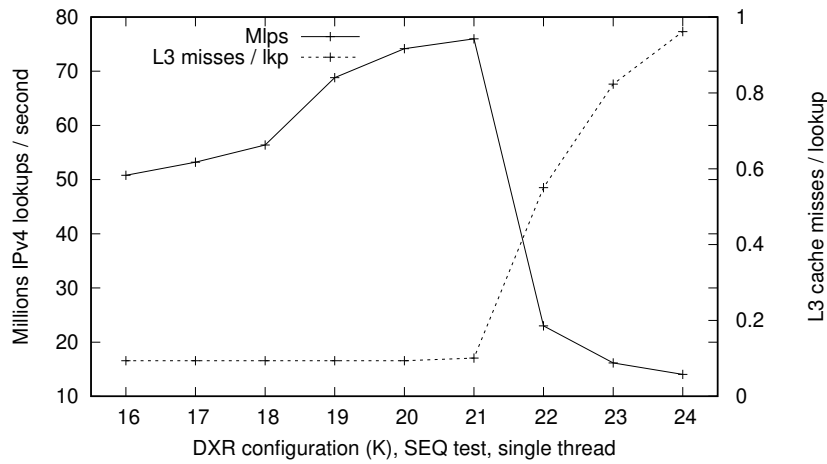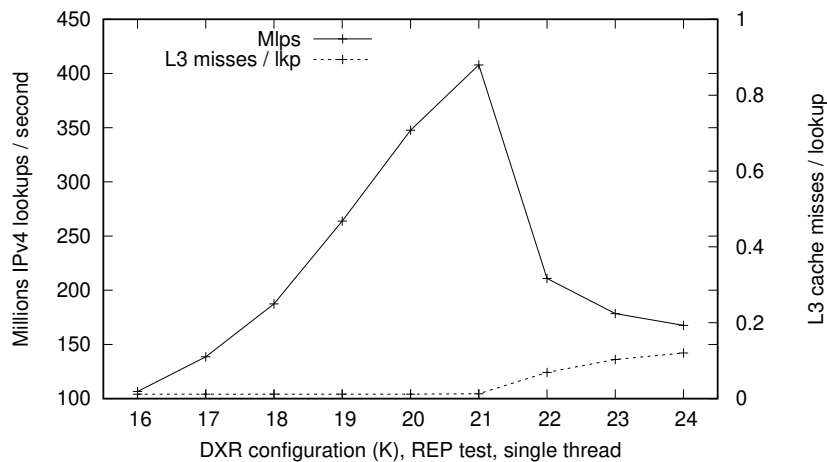

Figure 3.18: Single-thread lookup throughput with the corresponding L3 cache misses for a range of DXR configurations. RND test, UOREG 2018 snapshot, Intel i7-4771 CPU.

Figure 3.18 shows how the choice of DXR configuration (parameter K) impacts the single-thread lookup throughput and cache misses. On the particular machine used in this experiment, Intel i7-4771, as reported via the HWPMC infrastructure, the L3 cache miss rate is constant for configurations from $K = 16$ to $K = 20$, at a level of approximately 0.0939 misses per lookup. For $K = 21$ the miss rate rises only slightly to 0.0988 misses per lookup. Those L3 misses can be attributed to the influx of inbound keys (4 bytes) and to the stream of lookup results (2 bytes per lookup) being written to the main memory. Given that cache line size is 64 bytes wide, this in / out "overhead" amounts to 6/64, or 0.09375 cache line misses per lookup. Since the gap between the measured and computed miss rate is minimal, a conclusion follows that for con-

figurations up to $K = 21$, which also exhibits the top throughput at 211.18 Mlps, all references to lookup structures have been serviced from on-chip caches. Given that DXR total memory footprint for this particular configuration and FIB amounts to approximately 8.35 MBytes, and that the i7-4771 CPU is equipped with 8 MBytes of L3 cache, a conclusion can be drawn that the lookup algorithm efficiently utilizes the CPUs cache hierarchy, with negligible spilling to main memory.

For DXR configuration with $K = 22$, which requires approx. 16.08 Mbytes for the lookup structures, the cache miss rate bursts to 0.546 misses per lookup, and the lookup throughput collapses.

Figure 3.19: Single-thread lookup throughput with the corresponding L3 cache misses for a range of DXR configurations. SEQ test, UOREG 2018 snapshot, Intel i7-4771 CPU.

Figure 3.19 shows that while changing the request mode to serialized (SEQ) yields roughly threefold decrease in lookup throughput, cache miss rate per lookup remains unchanged for the same values of configuration parameter K.

Figure 3.20: Single-thread lookup throughput with the corresponding L3 cache misses for a range of DXR configurations. REP test, UOREG 2018 snapshot, Intel i7-4771 CPU.

Finally, when the same dataset is subjected to repeated lookups using the same key (REP test) in a 8-key sliding-window mode, a reduction in L3 cache misses compared to the previous two tests is observable, as shown in Figure 3.20. The baseline cache miss rate, as reported by HWPMC, is 0.0126 misses per lookup. Given that each 4-byte key is used 8 times, this corresponds to effectively 0.5 bytes of main memory loads per lookup, plus $2/8 = 0.25$ bytes of memory stores per lookup, which amounts to $0.75/64 = 0.0117$ cache line misses per lookup, very close to the actually recorded levels.
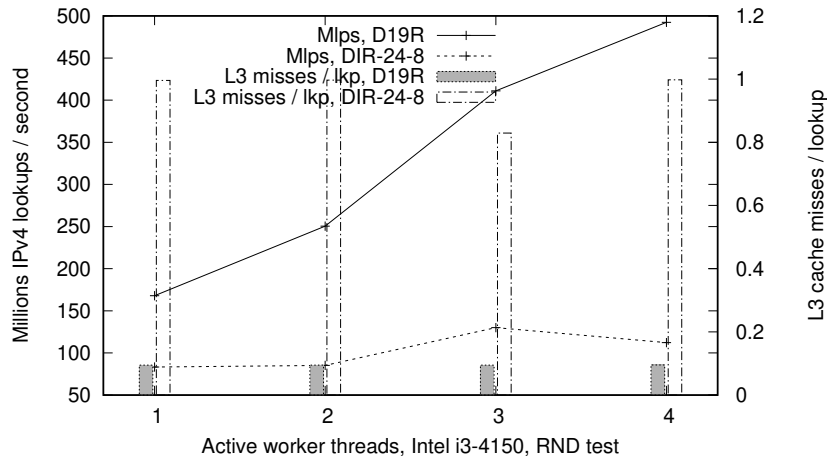
Figure 3.21: Lookup throughput and L3 cache misses for DXR with $K = 19$ and DIR-24-8. RND test, LINX 2017 snapshot, Intel i3-4150 CPU.
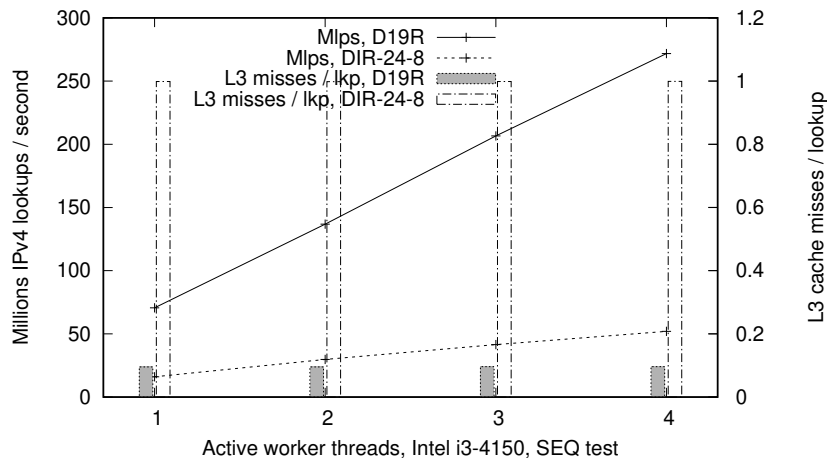
Figure 3.22: Lookup throughput and L3 cache misses for DXR with $K = 19$ and DIR-24-8. SEQ test (artificial dependencies between queries), LINX 2017 snapshot, Intel i3-4150 CPU.

In a followup experiment, Figure 3.21 shows the correlation between lookup throughput for DXR configured with $K = 19$ and DIR-24-8 on a low-end Intel CPU. Per statistics harvested from the HWPMC L3 miss counter, DIR-24-8 saturated the DRAM subsystem already with two worker threads at around 120 millions L3 cache misses per second, while achieving the top lookup throughput with three worker threads. In contrast to this, DXR scaled well to all
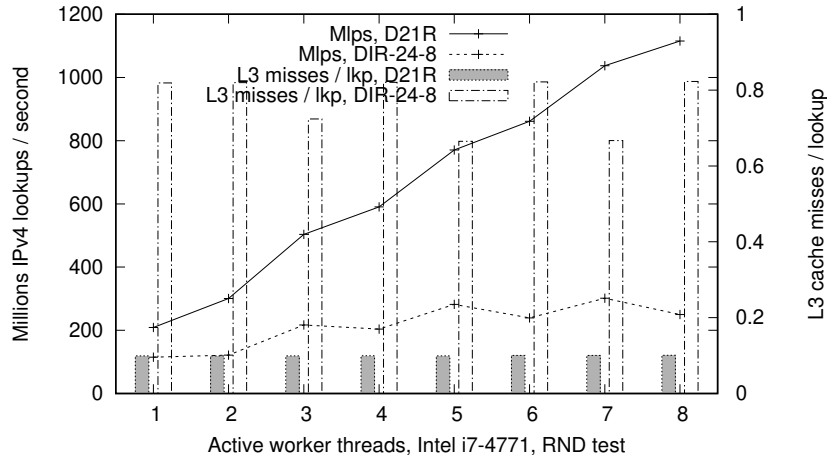
Figure 3.23: Lookup throughput and L3 cache misses for DXR with $K = 21$ and DIR-24-8. RND test, 2018 UOREG snapshot, Intel i7-4771 CPU.
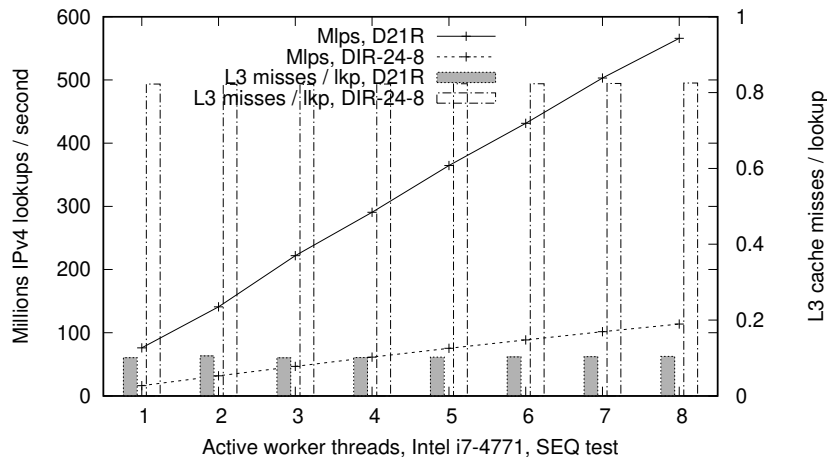
Figure 3.24: Lookup throughput and L3 cache misses for DXR with $K = 21$ and DIR-24-8. SEQ test (artificial dependencies between queries), 2018 UOREG snapshot, Intel i7-4771 CPU.

CPU cores, while achieving peak lookup throughput roughly four times higher than DIR-24-8, at a fraction of L3 cache misses. Again, a portion of L3 misses in all cases can be attributed to unavoidable fetching of random keys from the memory, as well as storing the results (next hop indices) back to another memory array. Figure 3.22 shows the similar effect on L3 cache trashing with the SEQ test which introduces artificial dependencies between successive lookups.

Figure 3.23 and Figure 3.24 show the results of the same experiment sequence repeated on another CPU, the Intel i7-4771. DXR outperforms DIR-24-8 by a factor of four when distributing lookup load over all 8 cores. A staircase-formed throughput pattern can be observed, as worker thread pairs are being scheduled on physical CPU cores which share two SMT execution contexts, and compete for processing units (ALU, memory interface), resulting in non-uniform throughput increases in case of DXR, and even slight decreases with DIR-24-8. An anomaly in L3 miss rates associated with DIR-24-8 are visible, as miss rates unexpectedly decrease for odd numbers of worker threads.
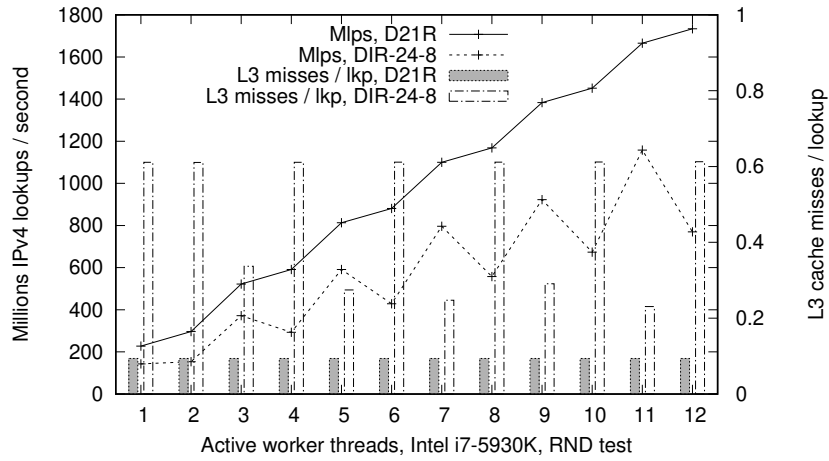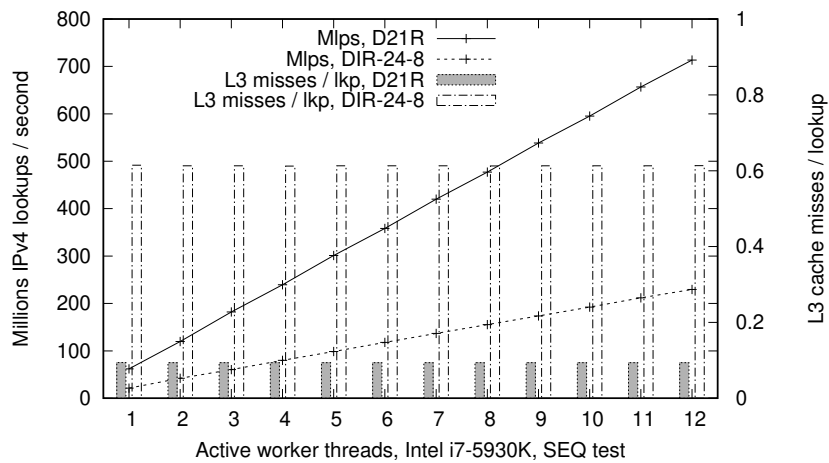
Figure 3.25: Lookup throughput and L3 cache misses for DXR with $K = 21$ and DIR-24-8. RND test, 2018 UOREG snapshot, Intel i7-5290K CPU.

Figure 3.26: Lookup throughput and L3 cache misses for DXR with $K = 21$ and DIR-24-8. SEQ test (artificial dependencies between queries), 2018 UOREG snapshot, Intel i7-5390K CPU.

Finally, Figure 3.25 and Figure 3.26 show the results of the same experiment sequence repeated on a more recent Intel i7-5390k processor. The staircase throughput pattern is even more pronounced here compared to the previous case, as is the anomaly of decreasing L3 miss rates with odd numbers of worker threads. The anomaly calls for further examination, which was not performed at this time.

Chapter 4

Further space and time optimizations

As shown in the previous chapter, DXR's memory footprint is reasonably compact for large and densely populated global routing tables (BGP), where the scheme offers a broad spectrum of practical space-speed configuration tradeoffs, depending on target application goals. However, when used with sparsely populated FIBs, such as in access routers or end hosts with static routing tables, typically consisting of only a few local networks and a default route, the scheme becomes highly inefficient in terms of required memory per prefix ratios. For example, with the best performing D21R scheme and a small static FIB with 5 prefixes, the lookup structures will occupy unreasonable 1.6 MBytes per prefix. This does not compare favorably compared to other LPM schemes, especially in private networks, where the number of prefixes is typically two to three orders of magnitude smaller compared to the global BGP view, and the prefixes are mostly constrained to a narrow fraction of the address space (e.g., 10.0.0./8).

Can a split of the direct lookup step into two stages improve DXR's spatial efficiency, without (significantly) sacrificing performance for the target (most challenging) application with full view BGP tables?

If such splitting of the single direct lookup step in multiple smaller ones could improve the spatial efficiency for sparsely populated tables, would it morph the DXR proposal into a standard multibit trie structure variant, which have been shown to exhibit LPM throughput levels inferior to DXR due to data dependencies and associated branches which have to be resolved while traversing through trie levels?

This section presents a proposal for spatial optimizations of DXR lookup structures, based on splitting the single direct stage in two, which are dubbed *Direct* and *eXtended* tables. An implementation of such a proposal is then subjected to a performance evaluation.

4.1 Data structures, deduplication

The space-optimized proposal builds upon the fact that DXR already performs leaf compression, i.e., identical range chunks are being deduplicated during the update process and are maintained as reference-counted objects, which was described previously in section 3.5. The hereby proposed extensions explore the idea that if the direct table gets split into two, the leaf blocks of the resulting intermediate table might also exhibit certain level of redundancy, and permit deduplication rates of practical value.

Using the existing DXR structures as a starting point, constructing two-level table structure is straingthforward. The address space previously covered by K bits is decomposed into $D + X$ bits, where $D + X = K$. A relatively small primary *direct* lookup table is now indexed by D leftmost bits of the lookup key, which holds indices to uniform blocks in the *extension* table, with each encompassing 2^X range table descriptors. 2 bytes suffice for storing indices in the *direct* table, if D is chosen such that $D \leq 16$. The middle X bits of the search key are used as an offset in the *extension* table block, which is then referenced to either resolve the lookup, or to proceed to a binary search in the *range* table chunk based on the remainder bits of the lookup key. The arrangement is show in in Figure 4.1.
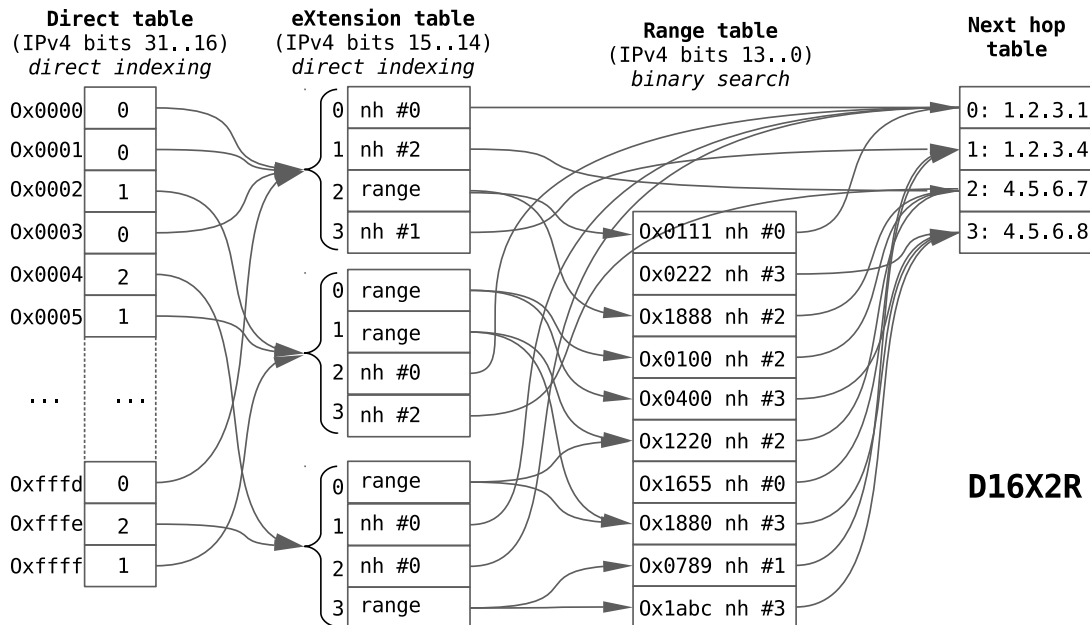

Figure 4.1: An example of lookup structure arrangement in the extended DXR scheme. In addition to range chunks which are already deduplicated, blocks of the *extension* table are also maintained as deduplicated, reference counted objects, permitting significant memory footprint savings compared to the original DXR's monolithic direct lookup table.

Since all the blocks in the *extension* table already have their references to the *range* table consolidated, identical blocks can be easily merged. Figure 4.2 reveals that the compression ratio is insignificant for configurations with $K = 16$, because range chunks are large, and therefore

range compression is ineffective, as previously shown in Figure 3.9. However, with $K = 21$, or more, the compression becomes more efficient, particularly with $D = 16$ (D16X*R). The memory savings can extend up to 70%.
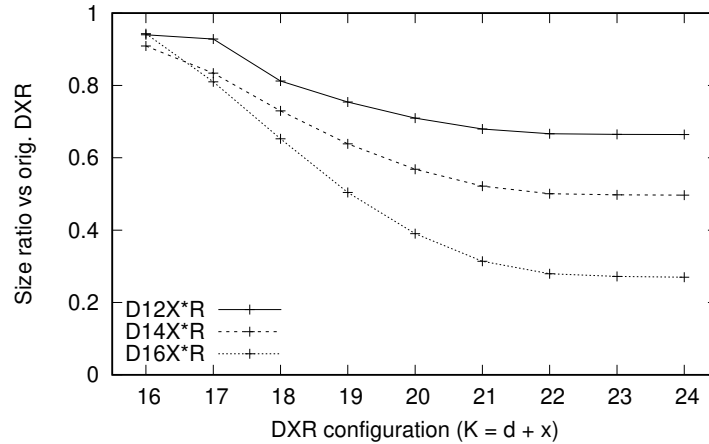

Figure 4.2: The ratio of the extended scheme's overall memory footprint compared the original DXR arrangement. D12X*R stands for $D = 12$, D14X*R for $D = 14$, and D16X*R for $D = 16$. $X = K - D$.

The almost four-fold improvement in memory footprint efficiency permits longer portion of the key to be used for *direct + extension* table indexing. It becomes feasible to choose values of K by one to two bits longer compared to the original DXR arrangement. Therefore the remaining binary search over address range is on average reduced by up to two iterations, which may compensate for the additional small *direct* table lookup step introduced over the original DXR scheme. As the performance evaluation presented in Section 4.3 will show, in certain configurations the scheme actually significantly speeds up the search process.

Deduplication of the *extension* table is based on generating a hash of each recomputed block, and comparing it against a table of already existing reference-counted blocks, and is therefore both trivial and fast. Thus, lookup structure updating speed remains virtually unaffected compared to the original DXR scheme.

Figures 4.3, 4.4 and 4.5 show the total memory footprint for various arrangements of parameters D and X, with D being fixed to 12, 14 and 16 bits respectively. The size of the range table remains exactly the same for all configurations with the same value of K, including the original unoptimized DXR arrangement, but the size of *direct* and *extension* lookup tables vary significantly.

Although more configuration choices for D and X than those presented in Figures 4.3, 4.4 and 4.5 are possible, 16 bits is the largest practical choice for parametar D, as it results with the *direct* table having a footprint of 128 KBytes, which presents a good fit for today's L2 CPU caches, varying in size from 256 go 512 KBytes. Increasing D to 17 or more would require widening the size of *direct* table entries from 16 to 32 bits, so a step from 16 to 17 bits for D would yield a four-fold expansion of *direct* table size, from 128 to 512 KBytes, i.e., double the
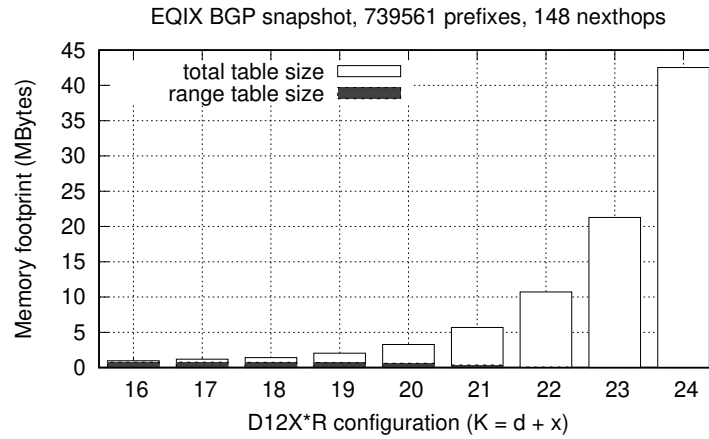
Figure 4.3: D12XR memory footprints for ascending values of *K*. *D* = 12, *X* = *K* - 12. EQIX 2018 FIB snapshot.
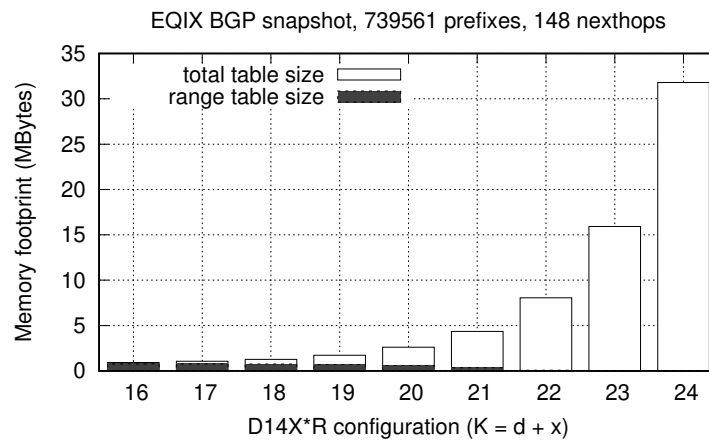

Figure 4.4: D14XR memory footprints for ascending values of *K*. *D* = 14, *X* = *K* - 14. EQIX 2018 FIB snapshot.

capacity of contemporary Intel CPU's L2 caches.

Lower values of *D* may present a better space/speed tradeoff for sparsely populated FIBs. For example, with *D* = 12, the size of *direct* table is only 8 KBytes. A small static FIB with 5 prefixes and D12X9R configuration (*D* = 12, *X* = 9, *K* = 21) will have the total memory footprint of all the lookup structures combined as low as 15 Kbytes. At 3 Kbytes per prefix this is over two orders of magnitude lower than the memory footprint of the original D21R arrangement, which operates with the same level of *K* = 21, and uses an identical range layout of *range* table.

4.2 Lookup algorithm

The lookup algorithm is straightforward as already outlined in the previous section: the smaller *direct* table indexed by *D* leftmost bits of the lookup key, which resolves a position of the next block in the *extension* table. The *extension* table block is indexed by the next *X* bits of
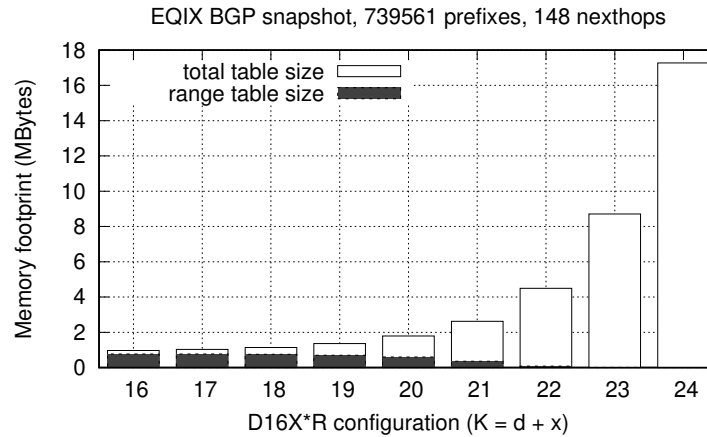
Figure 4.5: D16XR memory footprints for ascending values of K. $D = 16$, $X = K$ - 16. EQIX 2018 FIB snapshot.

the lookup key. If the referenced *extension* table entry points to the next hop, the lookup is complete. Otherwise, the (position, size) information in the entry select the portion of the range table on which to perform a binary search of the (remaining bits of the) key.

Resolving into the *extension* table always performed, i.e., there is no branching based on the readout from the *direct* table. This mandates two cache read accesses for all lookups, but permits OoO to pipeline the *direct* + *extension* memory fetch, and in our experiments has shown better performance than branching based on hits in the *direct* table.

Further speed gains have been achieved by precomputing a sliding window of *extension* table readouts, before going back several packets and checking the actual data, and either proceed with range lookup or be done at that point.

4.3 Performance evaluation

The same benchmarking methodology using streams of synthetic keys was used as described previously in section 3.6.

The first series of tests were aimed at determining single-thread throughput of several configurations of the extended DXR scheme, including a comparisson against the version presented in the previous chapter, which is included again in further benchmarks as a reference.

Figures 4.6, 4.7, and 4.8 show the observed throughput on a a machine equipped with the Intel i7-4771 CPU using a single worker thread subjected to random (RND), repeated (REP), and artificially serialized (SEQ) test patterns respectively, for the single-level direct table DXR, and for three modified variants configured with $D = 12$, $D = 14$, and $D = 16$.

As the number of bits (K) resolvable by *direct* and *extension* table lookups increase, all four DXR arrangements exhibit similar LPM throughput levels up to $K = 21$ in the RND test (Figure 4.6). With $K = 22$ the throughput of the reference version collapses, while the D16X6R

($D = 16$, $X = 6$) configuration reaches its top performance at 255.4 Mlps. The throuhput of D14X6R ($D = 14$, $X = 8$) peaks at 220.3 Mlps, while D12X10R drops slightly to 177.6 Mlps from the peak of 205.7 Mlps at D12X7R.
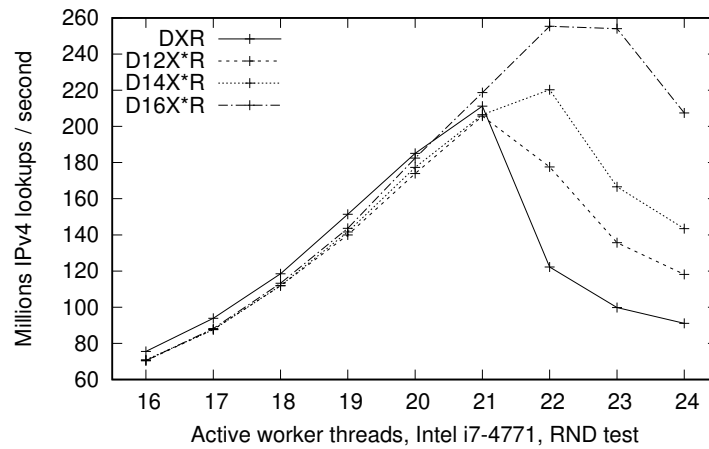

Figure 4.6: Single-thread lookup throughput comparisson between four base DXR variants configured with K = 16..24. RND test, UOREG 2018 snapshot, Intel i7-4771 CPU.

Figure 4.7 shows that throuhgputs in excess of nearly 400 Mlps are possible with locality in the stream of inbound keys, e.g., 397.0 Mlps with D16X6R and 409.5 with D21R. While those figures are of questionable real-world value, they are important as a reference for comparison against reports from other authors, who often lean towards benchmarking with focus on traffic traces, which cover only a fraction of the IPv4 address span.
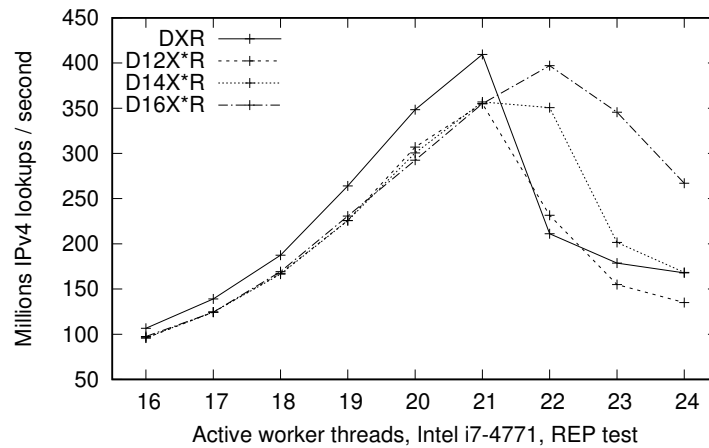

Figure 4.7: Single-thread lookup throughput comparisson between four base DXR variants configured with K = 16..24. REP test, UOREG 2018 snapshot, Intel i7-4771 CPU.

Figure 4.8 shows that even with artificial dependencies between the lookups, all DXR variants exceed 50 Mlps throughput per CPU core. The test is significant for applications where further decisions must be taken immediately based on the lookup result, such as in packet filters.

Figures 4.9, 4.10, and 4.11 show the single-thread throughput on AMD Ryzen 1700 CPU for RND, REP, and SEQ tests. The throughput peaks at 370 Mlps in the most significant test
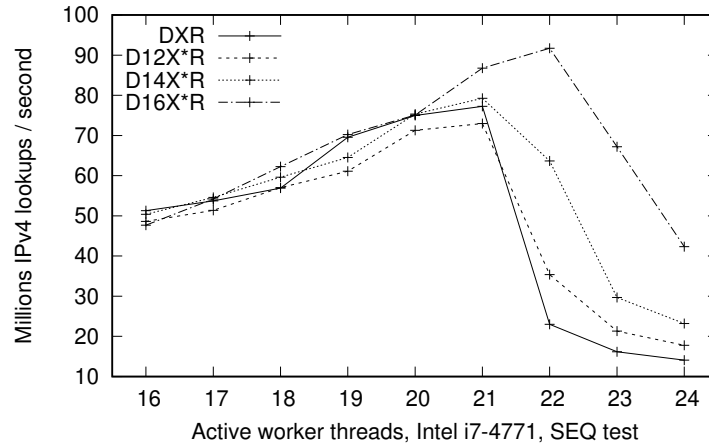
Figure 4.8: Single-thread lookup throughput comparisson between four base DXR variants configured with K = 16..24. SEQ test, UOREG 2018 snapshot, Intel i7-4771 CPU.
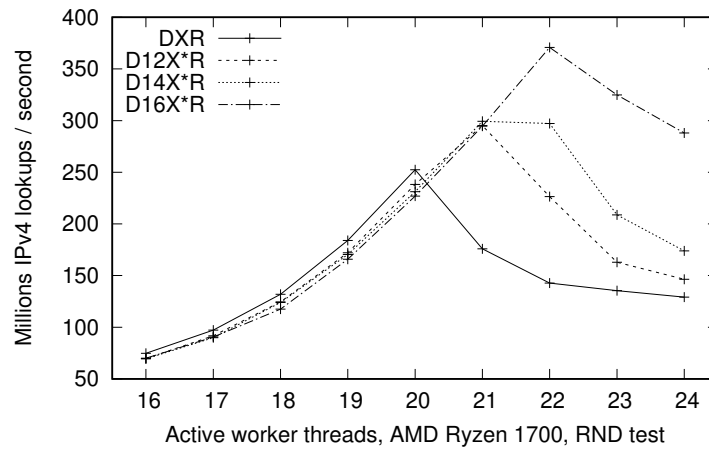

Figure 4.9: Single-thread lookup throughput comparisson between four base DXR variants configured with K = 16..24. RND test, UOREG 2018 snapshot, AMD Ryzen 1700 CPU.

(RND).

Figures 4.12, 4.13, and 4.14 show cache miss rates for extended DXR arrangement with D = 12, D = 14, and D = 16, with a single worker thread on an Intel i7-4771 CPU.

The subsequent figures show how the throughput of DXR variants scales on two AMD multi-core CPUs, the Ryzen 7-1700, and ThreadRipper 1950X. In all the tests the D16XR variant yields the best results, and all DXR variants significantly outperform the DIR-24-8 scheme, the throughput of which saturates quickly due to its high pressure on the main memory.
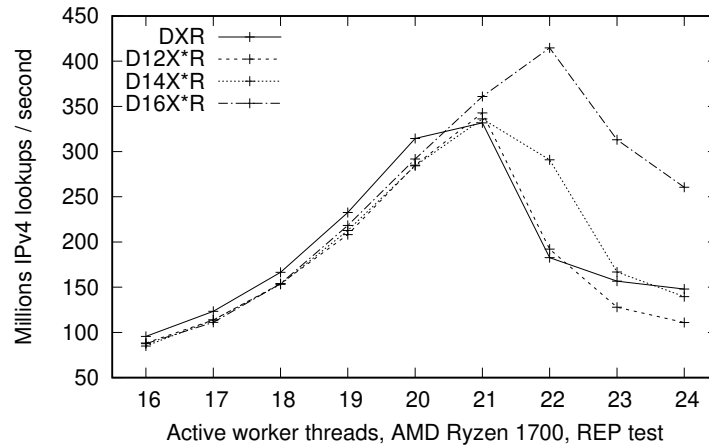
Figure 4.10: Single-thread lookup throughput comparisson between four base DXR variants configured with K = 16..24. REP test, UOREG 2018 snapshot, AMD Ryzen 1700 CPU.
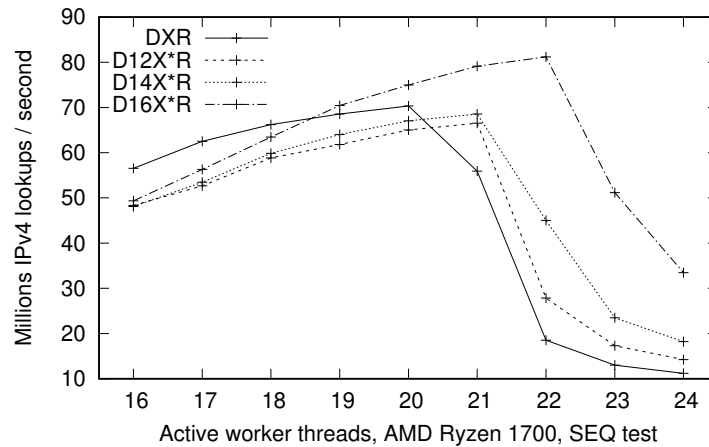

Figure 4.11: Single-thread lookup throughput comparisson between four base DXR variants configured with K = 16..24. SEQ test, UOREG 2018 snapshot, AMD Ryzen 1700 CPU.
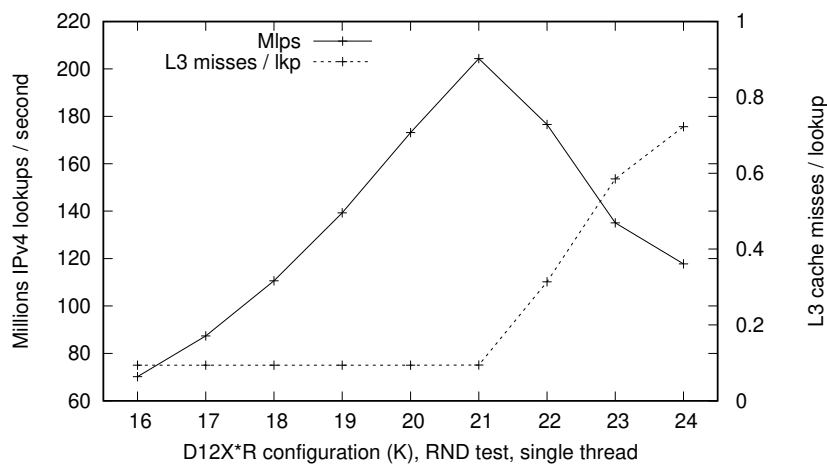

Figure 4.12: Single-thread lookup throughput with the corresponding L3 cache misses for a range of D12XR configurations. RND test, UOREG 2018 snapshot, Intel i7-4771 CPU.

Figure 4.13: Single-thread lookup throughput with the corresponding L3 cache misses for a range of D14XR configurations. RND test, UOREG 2018 snapshot, Intel i7-4771 CPU.
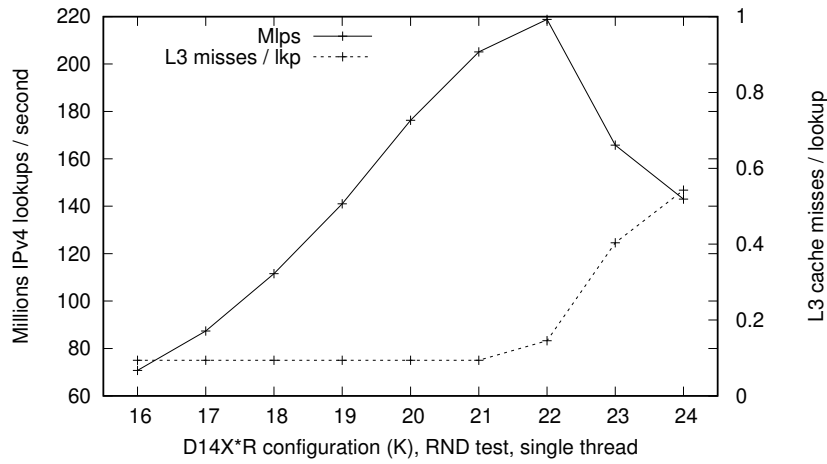

Figure 4.14: Single-thread lookup throughput with the corresponding L3 cache misses for a range of D16XR configurations. RND test, UOREG 2018 snapshot, Intel i7-4771 CPU.
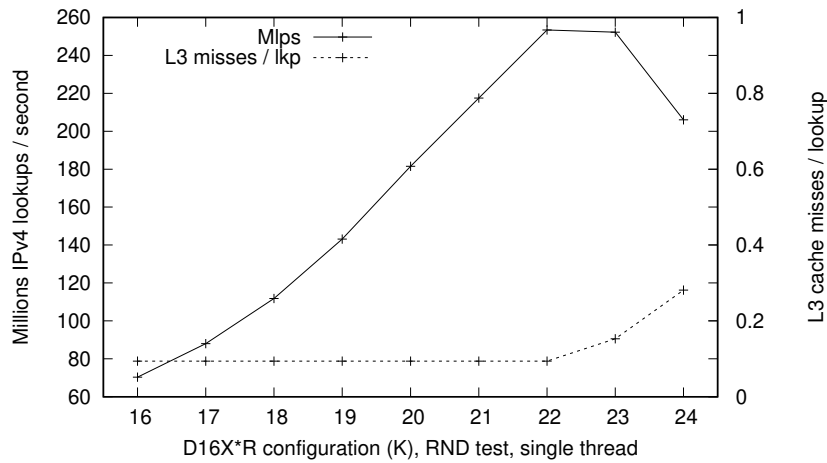

Figure 4.15: LPM throughput scaling with parallel worker threads. SEQ test (artificial dependencies between lookup iterations). EQIX 2018 snapshot, AMD Ryzen 1700 CPU

Figure 4.16: LPM throughput scaling with parallel worker threads. RND test (uniformly random keys). EQIX 2018 snapshot, AMD Ryzen 1700 CPU

Figure 4.17: LPM throughput scaling with parallel worker threads. REP test (simulated locality in traffic patterns). EQIX 2018 snapshot, AMD Ryzen 1700 CPU
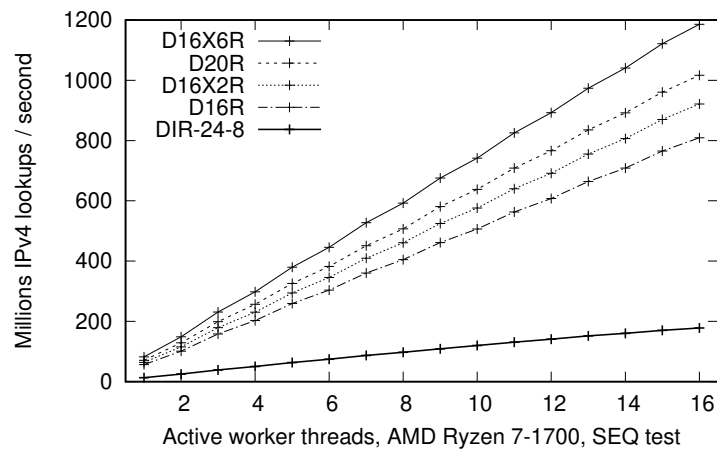

Figure 4.18: LPM throughput scaling with parallel worker threads. SEQ test (artificial dependencies between lookup iterations). EQIX 2018 snapshot, AMD ThreadRipper 1950X CPU
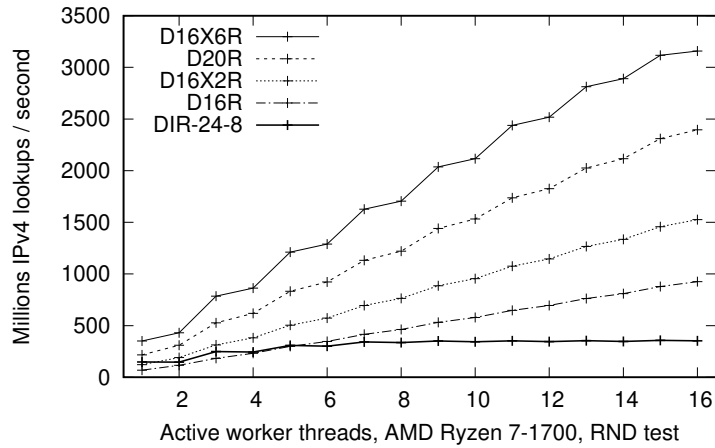
Figure 4.19: LPM throughput scaling with parallel worker threads. RND test (uniformly random keys). EQIX 2018 snapshot, AMD ThreadRipper 1950X CPU
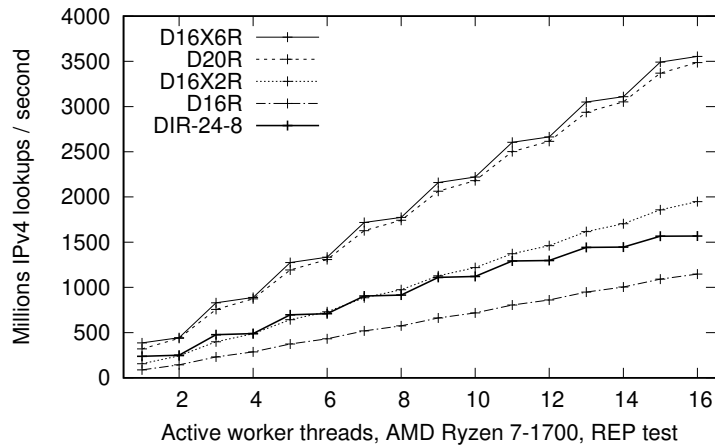

Figure 4.20: LPM throughput scaling with parallel worker threads. REP test (simulated locality in traffic patterns). EQIX 2018 snapshot, AMD ThreadRipper 1950X CPU
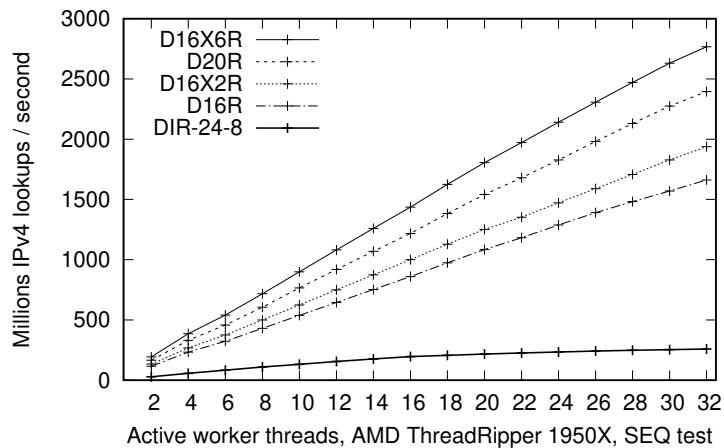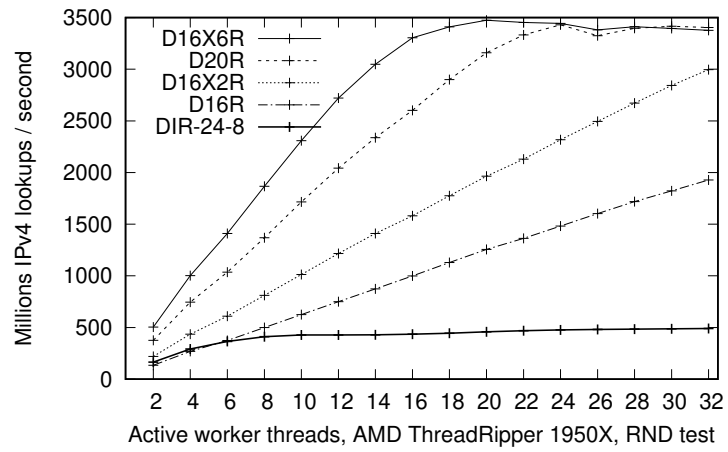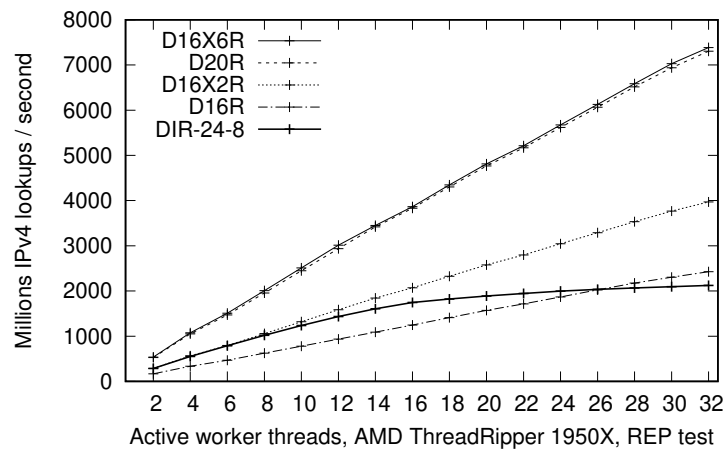
Chapter 5

Datapath integration

At minimum, a LPM implementation or library must support three elementary operations: insertion of a $\{prefix, label\}$ paire, deletion of a $prefix$, and finally a lookup function for a given key. Prototyping DXR started with implementing those operators, and gradually others were added. This chapter discusses the specifics of the author's efforts at integrating DXR in several conceptually different packet processing datapaths, each with a different application programming interface (API)s.

5.1 FreeBSD kernel

The initial DXR prototype [26] was implemented inside the FreeBSD kernel. The original motivation was driven by the observation that the standard BSD radix trie [18] was becoming the major bottleneck in packet forwarding applications with anything but trivial FIBs, and especially so when the FIB would correspond to a full-view BGP dataset of network prefixes. The goal was to provide a more efficient alternative to the standard FreeBSD LPM lookup function, in hope that this would make the OS feasible (again) for moderately fast Internet routing applications.

To retain compatibility with the existing routing protocol daemons, such as Quagga [55] or XORP [48], the obvious design choice was to retain all the existing userland to kernel routing APIs. As it was also obvious that DXR will need an auxiliary database from which the lookup structures will be derived, the already in-place and proven BSD radix trie was selected for that purpose. The existing BSD radix trie structure was extended with a single field which permitted keeping track of next hop data along each stored prefix in form of a small integer index in an additional reference-counted next hop table. This was implemented in addition to the (inefficient) standard practice in BSD to keep a full copy of next hop information along each stored network prefix, which was retained for compatibility with the existing tools and

applications.

As BSD routing socket API only provides methods for managing individual network prefixes, even partial recomputation of DXR's lookup structure on each prefix insertion or deletion was clearly not feasible, particularly with BGP databases consisting of several hundred thousands of prefixes. Therefore, deferred updating of lookup tables was implemented, which would be triggered following any change in the radix tree database, but only after the routing socket would remain idle for a predefined time interval (several milliseconds). This permitted multiple (thousands of) prefix insertions or deletions to be coalesced before updating the lookup structures, which is a process requiring the entire address space affected by the added or removed network prefixes to be traversed.

Finally, the DXR's lookup function was planted in the packet forwarding path, inside a modified version of the existing ip_fastforward() function. The choice of the LPM method could be made at run time, permitting tests to be conducted which confirmed that both the standard and the replacement function forwarded the packets in an identical way.

The primary value of the described development effort was that an initial implementation of DXR materialized, and that it could be shown to function correctly, by comparing the outcomes of the lookup process against a proven reference implementation, both using real traffic as well as synthetic (random) key streams.

However, the effort stopped short of achieving a real breakthrough in improving packet forwarding performance. Once the LPM bottleneck problem was solved, it became apparent that other non-trivial factors stood in the way of reaching packet forwarding rates in or above the 10 Mpps range. FreeBSD encapsulates packets in a structure called *mbuf*, which was designed when packet rates were miniscule by today's standards, while a more pressing issue was the amount of available memory for buffering packets. Therefore *mbufs* were devised to permit various modes of chaining smaller buffers, and over the years they accumulated a broad spectrum of auxiliary fields and flags targeting numerous specialized tasks, such as checksum flags of all kinds, or other transport-layer specific information. By the time device drivers properly populate and adjust all the required fields for each received packet, precious time is already irrevocably lost, even before any packet processing even begins. Once a device driver delivers the packet to the inbound part of the network stack, the software has to traverse over a plethora of flags and fields in extremely branchy sections of considerably complex code.

To compensate for the overhead of inefficient handing over each packet individually from device drivers to the network stack, an attempt was made to chain (group) multiple packets together before delivery to the next layer for further processing. However, the existing kernel code above the link layer was structurally composed with individual packet processing in focus, so that attempts at further improvements in the network layer were deemed futile and therefore stopped.

The whole effort was finally abandoned once a more viable generic framework for packet input / output (I/O) called netmap became available. The troubles with developing and debugging kernel-level code, compared to far more comfortable work in the user space which netmap offered, cemented the decision to abandon further efforts in improving FreeBSD kernel's packet forwarding datapath.

5.2   The Click Modular Router

Click [25] is a widely used software architecture for building flexible and configurable packet processing datapaths, which are assembled from libraries of modules (called *elements* in Click's parlance) using well-defined interfaces. Originally designed to run in the Linux kernel for efficient packet handoff, it may also work as a user space application, even more efficiently using netmap for packet I/O compared to the legacy kernel mode.

The simplicity of the already provided interfaces inside Click, combined with the platform's popularity motivated the author to (re)implement DXR as a Click module. Developing a custom Click processing module was a relatively straightforward task. The DXR bringup effort included porting the BSD radix tree code to Click as an auxilirary database, as previously mentioned in section 3.5.

The already available Click's routing table interfaces, such as the ability to atomically instantiate and populate large FIBs, greatly simplified experiemtation cycles, as almost no time was wasted for experiment setup, compared to relatively lengthy FIB setup procedures in FreeBSD when shell scripting was used for injecting routes into the kernel.

An optimized lookup function which operates on blocks (batches) of packets was also implemented, but since Click's traffic handoff interfaces were designed to operate on packet-by-packet basis only, the speed of performing LPM lookups over blocks of keys could be tested using synthetic load, since OoO execution mode makes it next to impossible to accurately and reliably capture the duration of extremely short individual code sequences.

Similarly to the first prototype developed for kernel-level operation in FreeBSD, the Click variant was shown to function correctly by comparing the outcomes of the lookup process against another LPM lookup Click module. Although the functional test with real traffic was successfull, benchmarking with real packets was not performed. Rather, the focus was put on implementing auxiliary handlers for controlling and monitoring benchmarking with synthetic key streams, such as selecting test parameters, preparing streams of random keys, triggering the benchmarks and collecting the results.

5.3   User-space Packet Processing Library

To enable embedding of DXR in standalone packet processing applications, the code from the Click prototype had to be backported from C++ to plain ANSI C. Again, BSD radix tree was retained as the auxiliary database for storing network prefixes.

The LPM cores of the two implementations (C++ / Click, and the standalone C library) are currently being kept in sync, but the difference is that the standalone library no longer keeps the reference counted table of next hop information, but rather passes the burden of that function to the specific application. Instead, the standalone DXR library expects the application to tag each prefix with a small integer label, which remains opaque to the library itself. This permits the application to interpret this label as it sees fit: the tag / label may be used for indexing a table of next hop information in router applications, while in traffic filtering scenarios the label may have different semantics.

The burden of populating and maintaining the network prefix database is also left entirely to the application, which furthermore must make the decision on when to update the lookup structures as a followup to any changes in the prefix database.

The small size of DXR's lookup structures permits for adopting an efficient synchronization strategy in which several independent versions of lookup structures may coexist at the same time. Once a new version of the structures is prepared, multiple worker threads may gradually switch from the old to using the new version, without having to block during the recomputation of lookup structures, or even being aware that an update is in progress. After all worker threads have signaled that they have switched over to using the new version, the old one may be safely deallocated. A minor complication with such a synchronization scheme is that labels which are attached to prefixes must remain valid over two consecutive versions of the lookup tables, but again, the burden of ensuring this constraint is met is left to the application.

The library was field-tested by having been incorporated in a packet processor application built on top of netmap [15] [56]. The application spawns multiple worker threads, each of which is assigned to servicing a single receive queue associated with a network interface. An additional thread is responsible for control and management tasks.

Load distribution over multiple queues / worker threads is performed in NIC hardware, beyond applications control. The NIC hardware uses an opaque hash function for distributing packets among processing cores, ensuring that packets belonging to a single traffic flow are always processed by the same core / thread.

Tests in a 10 GBit/s testbed have shown that zero-loss forwarding of packets with uniformly random source and destination addresses was possible at full line speed, i.e., at 14.88 Mpps, while performing DXR LPM lookups on each packet using a single CPU core. However, as the goals of the project which included the construction of a packet processing application built on

top of netmap and DXR were beyond the scope of this thesis, more specific results could not be disclosed at the time of this writing.

5.4   Future directions

Full 10 Gbps Ethernet line rate packet forwarding (14.88 Mpps) using a single CPU core can be sustained while doing DXR LPM lookups, even when running with reduced core frequency. Therefore, performing further experiments in a testbed equipped with faster (40G, 100G) interface cards would be justified for testing the practical limitations of the proposed LPM scheme.

Chapter 6

Related work

IP lookup algorithms have been well studied in the past, first for software-based solutions, and eventually focusing on designs that could be implemented in hardware to overcome the perceived (or actual) mismatch between network and CPU speeds. Ruiz-Sanchez et al. [57] and Waldvogel et al. [58] provide comprehensive surveys of software solutions up to the year 2001, which covers most of the research on software lookups. Gupta's thesis from 2000 [50] and Varghese book [51] from 2005 provide a more broader hardware / software insight, garnished with some anecdotal evidence from the industry. Therefore this section provides a brief summary only of the main techniques and proposals predating Varghese's book, along with an overview of selected more recently published works.

Traditional solutions involve tries [18], optimized to reduce the number of search steps by compressing long paths (Level-Compressed tries, [59]), or using n-ary branching (Multibit Tries, [60]). Given the small and fixed problem size, some ad-hoc solutions have been proposed that expand the root into a 2^k array of pointers to subtrees, as in DIR-24-8 [21] and in Lampson-Varghese [20]. Prefixes can be transformed into address ranges or intervals, which reduces the lookup to a binary search into an array of ranges [20]. Similarly exploiting the problem size, the Lulea scheme [19] partitions the trie in three levels (using 16, 8, 8 bits) and then uses a compact representation of the pointers.

As an alternative, Waldvogel et al. [58] propose the use of separate hash tables for each prefix length, starting the search from the most specific prefix and then moving up. This scheme is elegant but not particularly fast compared to other solutions for IPv4.

Caching recent lookup results using on-chip memory is discussed for instance in [61] and [62]. Chiueh and Pradhan achieved around 88 Mlps with host address caching on a 500 MHz DEC Alpha with 1 Mbyte of L2 cache (updates were not discussed) in 1999 [61]. The approach presented in [62] relies on temporal locality in the lookups, which is frequent in the leaves but less so in the core of the network.

Especially important in [57] is the comparison of actual run times of multiple algorithms,

which permits ranking them irrespective of absolute performance. The peak performance reported in the literature for such software solutions ranges between 2 and 5 Mlps on 1999 machines [57], and 3 to 20 Mlps on 2006 hardware [63].

Scaling these figures to modern hardware is not trivial, because the performance is dominated by memory access latencies. In fact, all the rest being equal, performance may vary by an order of magnitude or more depending on routing table size and request distributions. This also means that the memory footprint of a lookup scheme has a strong impact on its feasibility, especially as the number of prefixes grows (going from approx. 38 K prefixes in 1997 to the current 760 K prefixes in a full BGP table). In this respect, existing schemes tend to have quite large memory footprints, from the 24 bytes per prefix of the Lampson-Varghese scheme [20] to the 4.5 bytes per prefix of the Lulea [19].

The problem size can be reduced by performing routing table aggregations. SMALTA [64] shows a practical, near-optimal FIB aggregation scheme that shrinks the forwarding table size without modifying routing semantics or the external behavior of routers, and without requiring changes to FIB lookup algorithms and associated hardware and software. The claimed storage reduction is by at least 50%.

Due to the general inability of performing packet processing at line rates in software, a shift of interest towards hardware-based solutions for routing lookups was evident over the past decade and a half. As mentioned in the previous section, however, the combination of faster processing nodes, and an increased interest in virtualization, makes software IP lookups relevant again.

These performance numbers were/are not adequate for multi Gbit software routers, especially considering that the route lookup is only one of the many operations that must be performed on incoming traffic, hence may consume only a fraction of the total CPU time avalable for packet processing.

Therefore a general shift of interest ocured towards solutions that are suitable to efficient hardware implementation. Shape Shifting Trie proposed in 2005 [65] claims wire-speed processing for OC192 link using a single quad data rate II (QDRII) static random access memory (SRAM) chip using seven data structure accesses for route tables with more than 150,000 IPv6 prefixes. In 2006 Leu and Chang [66] proposed a lookup algorithm for IPv4 and IPv6. The main advantage is that time complexity is not related to the length of IP addresses, as for LC-Trie, Binary Search Architecture and Asymmetric-Tree schemes, so it can be applied on IPv6 addresses. There are only some simulation results presented and the reported performance is modest, from $400\mu s$ (best) to $700\mu s$ per lookup.

Alternatives to the use of a trie do exist. The Lulea scheme [19] does binary search on the number of prefix lengths. Lampson et al. [20] propose expanding prefixes into ranges which can be looked up with a binary search scheme. The whole set is partitioned in 2^{16} subset, using

the first 16 bits as an index to reach the correct subset. Reported performance is similar to that of the Lulea scheme [19], and between 2 and 10 times faster than the BSD scheme on the same hardware. Compared to the DXR scheme, [20] has a large memory footprint (700 Kbytes for just 38 K prefixes in 1999) which does not scale well with the 740 K prefixes of today's table.

There are a lot of hardware based implementations, such as [67] that uses hash-based membership query to limit off-chip memory accesses per lookup to one and to balance memory utilization among the memory modules (using a data structure called Prefix-Compressed Trie that reduces the size of a bitmap by more than 80%). The achieved simulation and implementation results [67] show that FlashTrie can achieve 160-Gbps worst-case throughput while simultaneously supporting 2-M prefixes for IPv4 and 279-k prefixes for IPv6 using one FPGA chip and four DDR3 synchronous dynamic random access memory (SDRAM) chips. FlashTrie also supports incremental real-time updates.

Another approach to fast IP lookups is the use of memory pipelines as proposed in [68], [69] and [70]. A long pipeline is used to produce one lookup in every clock cycle. In 1998 [21] reported 20×10^6 lookups per second implemented in hardware, using pipelined architecture with 50ns DRAM.

Early IP routers were all entirely software-based. Since by today's standards both line speeds and routing tables were miniscule, this worked well until mid-1990s when the Internet begun to expand at unprecedented rates. A wider adoption of faster transmission technologies, such as 155 Mbit/s Asynchronous Transfer Mode (ATM) or 100 Mbit/s Ethernet, along with rapid increases in global routing table sizes and the introduction of CIDR [2] pushed software routers to their limits and called for rapid innovations.

None of the software-based proposals could keep up with the exponential growth of both transmission speeds (1 and 10 Gbit/s) and the global routing table size, which by 1997 included over 40,000 prefixes. The schemes had quite large memory footprints, from 24 bytes per prefix of the Lampson-Varghese scheme [20] to 4.5 bytes per prefix of the Lulea [19], which prevented the lookup structures to fit into CPU caches as BGP table sizes continued to grow.

Both the research community and the industry eventually shifted their focus to routing lookup methods optimized for dedicated hardware. Early implementations were constructed around ternary content-addressable memory (TCAM) [71], but again those could not keep up with BGP table increases due to TCAM's low density and high power dissipation [22].

To cope with unabated BGP table growth, proposals to cache recent lookups in small but fast on-chip memories have surfaced occasionally (such as [61] or [62]) but never got embraced since both the vendors and operators learned that betting on traffic locality does not work well inside the Internet core due to unpredictable and constantly evolving traffic patterns.

A class of hardware-optimized approaches expands the root of the tree into a 2^k array of pointers to sub trees, such as DIR-24-8 [21] which could yield around 20 Mlps using a pipelined

ASIC- or FPGA-based implementation and two external commodity DRAM chips. As more throughput could be achieved by simply throwing more parallel hardware (DRAM chips) at the task, the major router vendors have been reportedly taking that route [72] to scale their ASICs into 100-300 Mlps throughput range, but cannot scale much further.

Recent proposals shift the focus back to CPUs for solving the problem of fast routing lookups. This author's initial DXR proposal [26] from 2012 reported compact FIB encoding from 1.8 Bytes per IPv4 prefix, and over 700 Mlps on an 8-core commodity CPUs.

Retvari et al. [73] [29] propose an information-theoretic approach for FIB compression to less than a byte per prefix, and projected lookup speeds to around 18 Mlps per CPU core for a FIB dataset of 440.000 prefixes. A recent derivate of their approach goes a step further by proposing lossy compression [74] of LPM structure, which is of questionable practical value, particularly in software, but also with TCAMs which that proposal puts in focus. Nevertheless, despite the reported lookup throughputs being an order of magnitude lower compared to other contemporary schemes, Retvari's et al. contribution in achieving high compression rates is significant. Their proposal for detecting and reducing redundancies in leaf levels of the tree encouraged the quest for appropriate techniques which could be applied to improving DXR's space / speed tradeoffs.

Yang et al. propose SAIL [31] [75], claiming LPM throughputs of 236 Mlps with random traffic and 625 Mlps with localities in traffic patterns. Their scheme is based on four-level multibit trie, of which the majority of lookups can be resolved by one access per each of the first three tables. The fourth serves as an overflow table for (rare) prefixes more specific than /24, similarly to the DIR-24-8 scheme. The authors elaborate that the first two levels have memory footprint bounded to 2.13 MBytes, but the sizes of the subsequent two tables cannot be deduced from their report, besides that than those are considerably larger than the previous two. Independent reviewers [30] report memory footprints in excess of 40 MBytes for SAIL. The source code which the authors made publicly available reveals that inside their test loop all lookup results are simply discarded. This not only unrealistically reduces the pressure on the memory subsystem (results never go to the main memory), but the OoO machinery can clobber (i.e., discard) the previous lookup result even before it gets completely resolved, by writing the next (partially resolved) result over the same CPUs architectural register.

Asai and Ohara propose Poptrie [30], which reportedly peaks between 174 and 240 Mlps with a single core and tables with 500-800k routes, and can achieve 914 Mlps with four CPU cores. Poptrie is an extension of a multiway trie, and splits the lookup tables into two, one for internal and the other for leaf nodes. Lookup key is processed 6 bits at a time, or alternatively in a single direct lookup of a wider stride. Similarly to DXR, Poptrie performs merger of a set of prefixes with the identical nex hop that belong to a subtree without any gap. Overall, in terms of memory footprint and reported performance, per Asai's report Poptrie offers similar

throughputs and space / speed tradeoffs to DXR.

Chapter 7

Conclusion

The author's early DXR proposal, which forms the basis for this thesis, was among the first recent impulses which prompted the research community to begin reviewing its (di)stance towards software routing lookups, which were perceived as a completely lost cause for almost a decade and half. The field become active again, with several other practical and efficient proposals emerging over the past few years.

During the interval from when the original proposal was published (late 2012) to the time when this thesis is being submitted (early 2019) the size of the global BGP routing database almost doubled, from 417,000 to over 760,000 prefixes, and the number is still growing. The proposed hybrid direct / range LPM lookup concept not only stood the test of time by easily absorbing this unabating BGP table inflation, but thanks to the refinements first published in this thesis, its performance grew almost five-fold, from approximately 700 Mlps to 3.5 billion routing lookups per second (Glps) in synthetic tests when subjected to streams of random lookup keys, and exceeding 7 Glps with locality in key streams. Much of the mentioned performance gain is due to improved ILP and cache efficiency in contemporary CPUs compared to their counterparts from 2012, although the most significant contributor is the increasing number of available processing cores. DXR makes efficient use of parallel processing units with near linear throughput scaling, which raises expectations that further performance gains might be obtainable on future microprocessor platforms with even bigger processing core counts.

An area left for future work is exploring the potential of NUMA architectures. A full throughput saturation was observed when scheduling LPM test threads on the second half of an inherently NUMA CPU, the AMD ThreadRipper. Strategies for better scaling in NUMA topologies, such as replicating lookup structures to physical memory blocks local to each NUMA node, as well as allocating per-node memory blocks for both inbound keys and the lookup results, should be further developed and evaluated.

Nevertheless, to the best of the author's knowledge, the 3.5 Glps throughput level puts DXR far ahead of all LPM (software) lookup proposals published to this date. The achieved

throughput is more than an order of magnitude higher than the capacity of a state-of-the-art router ASIC, the Cisco nPower X1, and almost two orders of magnitude faster than the most recent proposal based on multiway range LPM search implemented in an FPGA.

Moreover, among the recent LPM lookup proposals with throughputs which break the 50 Mlps per CPU core barrier, DXR has the lowest memory footprint, only 1.32 bytes per IPv4 prefix in the most compact configuration. Other proposals with even more compact FIB encoding (below 0.5 bytes per prefix) have emerged recently, but their lookup performance significantly lags behind (by more than an order of magnitude), because of overly branchy code with lengthy iterations required for LPM resolution over the compressed data structures. Two of the other recent proposals for fast LPM in software, SAIL and PopTrie, have so far not been demonstrated to scale as far as DXR on general-purpose CPUs with large number of processing cores.

The fact that the direct / range LPM proposal strikes a useful balance between high lookup throughputs and reasonable sizes of its data structures makes it a practical, viable option for application in today's virtualized software packet datapaths, where several routing contexts compete not only for CPU cycles, but even more so for cache space. The throughput / footprint balance is equally important in applications where lookups over multiple tables have to be performed on per-packet basis in a single datapath.

An implementation of DXR as a C library was integrated in an experimental datapath and subsequently field-tested by forwarding all departmental IPv4 traffic at Gbps speeds for several months. While the primary goal of that particular experiment was beyond the scope of this thesis, it indicates DXR's suitabilty for practical application in real-life, robust packet processors. Experiments in a 10 Gbps Ethernet testbed have shown that a packet datapath built on top of netmap and DXR sustains line rate throughput (14.88 Mpps) while performing LPM lookups using a single CPU core, even when running with reduced core frequency.

Much of the performance and scaling potential of DXR comes from the simplicity of the algorithm and the small size of its data structures, especially in the variant which peforms leaf node deduplication on the intermediate (extension) lookup table. At its core, this thesis resurrects several concepts well-known but old, thus mostly abandoned as overly simple and obsolete, and applies them effectively to a contemporary problem and contemporary general-purpose computing hardware.

Bibliography

[1] Coffman, K. G., Odlyzko, A. M., "Internet growth: Is there a "Moore's Law" for data traffic?", in Handbook of massive data sets. Springer, 2002, pages 47–93.

[2] Fuller, V., Li, T., Yu, J., Varadhan, K., "Classless inter-domain routing (CIDR): an address assignment and aggregation strategy", Tech. Rep., 1993.

[3] Cittadini, L., Muhlbauer, W., Uhlig, S., Bush, R., Francois, P., Maennel, O., "Evolution of internet address space deaggregation: myths and reality", IEEE Journal on Selected Areas in Communications, Vol. 28, No. 8, 2010, pages 1238–1249.

[4] Meyer, D., Zhang, L., Fall, K., "Report from the IAB workshop on routing and addressing", Internet Requests for Comments, RFC 4984, September 2007, available from: http://www.rfc-editor.org/rfc/rfc4984.txt

[5] Huston, G., "What's so special about 512?", Internet Protocol Journal, Vol. 17, No. 2, 2014, pages 2–18.

[6] Edwards, C., "Internet routing failures bring architecture change back to the table", ACM News, 2014, available from: http://cacm.acm.org/news/178293-internet-routing-failures-bring-architecture-changes-back-to-the-table/fulltext

[7] Wobker, L., "Evolution of core routing hardware and software", CiscoLive, 2014, available from: https://www.alcatron.net/Cisco%20Live%202014%20Melbourne/Cisco%20Live%20Content/Service%20Provider/BRKSPG-2640%20%20Evolution%20of%20Core%20Routing%20Hardware%20and%20Software.pdf

[8] Zec, M., Mikuc, M., "Real-time ip network simulation at gigabit data rates", in Proc. International Conference on Telecommunications (ConTEL), Zagreb, Croatia. Citeseer, 2003.

[9] Salopek, D., Vasić, V., Zec, M., Mikuc, M., Vašarević, M., Končar, V., "A network testbed for commercial telecommunications product testing", in Software, Telecommunications and Computer Networks (SoftCOM), 2014 22nd International Conference on. IEEE, 2014, pages 372–377.

[10] Bianco, A., Birke, R., Bolognesi, D., Finochietto, J. M., Galante, G., Mellia, M., Prashant, M., Neri, F., "Click vs. Linux: two efficient open-source IP network stacks for software routers", in High performance switching and routing, 2005. HPSR. 2005 workshop on. IEEE, 2005, pages 18–23.

[11] Bolla, R., Bruschi, R., "The IP lookup mechanism in a Linux Software Router: performance evaluation and optimizations", in High Performance Switching and Routing, 2007. HPSR'07. Workshop on. IEEE, 2007, pages 1–6.

[12] Brouer, J. D., "Network stack challenges at increasing speeds", in Proceedings of the Linux Conference, Auckland, New Zealand, 2015, pages 12–16.

[13] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., Ratnasamy, S., "RouteBricks: Exploiting parallelism to scale software routers", in SOSP '09. Big Sky, Montana, USA: ACM, 2009, pages 15–28, available from: http://doi.acm.org/10.1145/1629575.1629578

[14] Han, S., Jang, K., Park, K., Moon, S., "PacketShader: a GPU-accelerated software router", in SIGCOMM '10. ACM, 2010, pages 195–206.

[15] Rizzo, L., "netmap: A novel framework for fast packet I/O", in Usenix ATC 2012. Usenix, 2012.

[16] Baker, F., "Requirements for IP version 4 routers", RFC 1812, 1995.

[17] Rekhter, Y., Li, T., Hares, S., "A border gateway protocol 4 (BGP-4)", RFC 4271, 2006.

[18] Sklower, K., "A tree-based packet routing table for Berkeley Unix", in USENIX Winter Conference, 1991, pages 93-104.

[19] Degermark, M., Brodnik, A., Carlsson, S., Pink, S., "Small forwarding tables for fast routing lookups", SIGCOMM Computer Communication Review, Vol. 27, No. 4, Oct. 1997, pages 3–14, available from: http://doi.acm.org/10.1145/263109.263133

[20] Lampson, B., Srinivasan, V., Varghese, G., "IP lookups using multiway and multicolumn search", IEEE/ACM Trans. on Networking, 1998, pages 324–334.

[21] Gupta, P., Lin, S., McKeown, N., "Routing lookups in hardware at memory access speeds", in INFOCOM, 1998, pages 1240-1247.

[22] Zane, F., Narlikar, G., Basu, A., "CoolCAMs: Power-efficient TCAMs for forwarding engines", in INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies, Vol. 1. IEEE, 2003, pages 42–52.

[23] Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., Uhlig, S., "Software-defined networking: A comprehensive survey", Proceedings of the IEEE, Vol. 103, No. 1, 2015, pages 14–76.

[24] Naous, J., Gibb, G., Bolouki, S., McKeown, N., "NetFPGA: reusable router architecture for experimental research", in Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow. ACM, 2008, pages 1–7.

[25] Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M. F., "The click modular router", ACM Transactions on Computer Systems (TOCS), Vol. 18, No. 3, 2000, pages 263–297.

[26] Zec, M., Rizzo, L., Mikuc, M., "DXR: towards a billion routing lookups per second in software", ACM SIGCOMM Computer Communication Review, Vol. 42, No. 5, 2012, pages 29–36.

[27] Zec, M., Mikuc, M., "Pushing the envelope: Beyond two billion IP routing lookups per second on commodity CPUs", in Software, Telecommunications and Computer Networks (SoftCOM), 2017 25th International Conference on. IEEE, 2017, pages 1–6.

[28] Huston, G., "BGP routing table analysis reports", 2018, available from: http://bgp.potaroo.net/

[29] Rétvári, G., Tapolcai, J., Kőrös i, A., Majdán, A., Heszberger, Z., "Compressing IP forwarding tables: towards entropy bounds and beyond", IEEE/ACM Transactions on Networking, Vol. 24, No. 1, 2016, pages 149–162.

[30] Asai, H., Ohara, Y., "Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup", in ACM SIGCOMM Computer Communication Review, Vol. 45, No. 4. ACM, 2015, pages 57–70.

[31] Yang, T., Xie, G., Li, Y., Fu, Q., Liu, A. l. X., Li, Q., Mathy, L., "Guarantee IP lookup performance with FIB explosion", ACM SIGCOMM Computer Communication Review, Vol. 44, No. 4, 2015, pages 39–50.

[32] Mack, C. A., "Fifty years of Moore's law", IEEE Transactions on semiconductor manufacturing, Vol. 24, No. 2, 2011, pages 202–207.

[33] Waldrop, M. M., "The chips are down for Moore's law.", Nature, Vol. 530, No. 7589, 2016, pages 144–147.

[34] Danowitz, A., Kelley, K., Mao, J., Stevenson, J. P., Horowitz, M., "CPU DB: recording microprocessor history", Communications of the ACM, Vol. 55, No. 4, 2012, pages 55–63.

[35] Calhoun, B. H., Cao, Y., Li, X., Mai, K., Pileggi, L. T., Rutenbar, R. A., Shepard, K. L., "Digital circuit design challenges and opportunities in the era of nanoscale CMOS", Proceedings of the IEEE, Vol. 96, No. 2, 2008, pages 343–365.

[36] Intel, "Intel 64 and ia-32 architectures optimization reference manual", 2016.

[37] Butler, M., Yeh, T.-Y., Patt, Y., Alsup, M., Scales, H., Shebanow, M., "Single instruction stream parallelism is greater than two", in ACM SIGARCH Computer Architecture News, Vol. 19, No. 3. ACM, 1991, pages 276–286.

[38] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D. et al., "Meltdown: Reading kernel memory from user space", in 27th USENIX Security Symposium (USENIX Security 18), 2018, pages 973–990.

[39] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y., "Spectre attacks: Exploiting speculative execution", arXiv preprint arXiv:1801.01203, 2018.

[40] Gras, B., Razavi, K., Bos, H., Giuffrida, C., "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks", in 27th USENIX Security Symposium (USENIX Security 18), 2018, pages 955–972.

[41] Molka, D., Hackenberg, D., Schöne, R., Nagel, W. E., "Cache coherence protocol and memory performance of the Intel Haswell-ep architecture", in Parallel Processing (ICPP), 2015 44th International Conference on. IEEE, 2015, pages 739–748.

[42] Wu, Z. P., Krish, Y., Pellizzoni, R., "Worst case analysis of DRAM latency in multi-requestor systems", in Real-Time Systems Symposium (RTSS), 2013 IEEE 34th. IEEE, 2013, pages 372–383.

[43] Tullsen, D. M., Brown, J. A., "Handling long-latency loads in a simultaneous multithreading processor", in Microarchitecture, 2001. MICRO-34. Proceedings. 34th ACM/IEEE International Symposium on. IEEE, 2001, pages 318–327.

[44] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., Kohler, E., "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors", ACM Transactions on Computer Systems (TOCS), Vol. 32, No. 4, 2015, page 10.

[45] Intel, D., "Intel Data Plane Development Kit", Programmer's Guide–http://dpdk. org/-doc/guides/prog_guide, 2016.

[46] Navarro, J., Iyer, S., Druschel, P., Cox, A., "Practical, transparent operating system support for superpages", ACM SIGOPS Operating Systems Review, Vol. 36, No. SI, 2002, pages 89–104.

[47] Zec, M., "Implementing a Clonable Network Stack in the FreeBSD Kernel", in USENIX Annual Technical Conference, FREENIX Track, 2003, pages 137–150.

[48] Handley, M., Kohler, E., Ghosh, A., Hodson, O., Radoslavov, P., "Designing extensible IP router software", in Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association, 2005, pages 189–202.

[49] Morrison, D. R., "PATRICIA-practical algorithm to retrieve information coded in alphanumeric", Journal of the ACM (JACM), Vol. 15, No. 4, 1968, pages 514–534.

[50] Gupta, P., Mckeown, N. W., Algorithms for routing lookups and packet classification. Stanford University Diss, 2000.

[51] Varghese, G., Network Algorithmics: An Interdisciplinary Approach To Designing Fast Networked Devices, ser. The Morgan Kaufmann Series in Networking. Elsevier/Morgan Kaufmann, 2005, available from: http://books.google.hr/books?id=01QORuRF6fIC

[52] Suri, S., Varghese, G., Warkhede, P. R., "Multiway range trees: Scalable IP lookup with fast updates", in Proc. IEEE GLOBECOM '01 , v3 2001, 2001, pages 1610–1614.

[53] "University of Oregon RouteViews project", Eugene, OR.[Online]. Available: http://www.routeviews.org.

[54] Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., Francillon, A., "Reverse engineering Intel last-level cache complex addressing using performance counters", in International Workshop on Recent Advances in Intrusion Detection. Springer, 2015, pages 48–65.

[55] Jakma, P., Lamparter, D., "Introduction to the quagga routing suite.", IEEE Network, Vol. 28, No. 2, 2014, pages 42–48.

[56] Rizzo, L., "Revisiting network I/O APIs: the netmap framework", Comm. of the ACM, Vol. 55, No. 3, 2012, pages 45–51.

[57] Ruiz-Sanchez, M., Biersack, E. W., Dabbous, W., "Survey and taxonomy of IP address lookup algorithms", IEEE Network, Vol. 15, 2001, pages 8–23.

[58] Waldvogel, M., Varghese, G., Turner, J., Plattner, B., "Scalable high-speed prefix matching", ACM Trans. Comput. Syst., Vol. 19, No. 4, Nov. 2001, pages 440–482, available from: http://doi.acm.org/10.1145/502912.502914

[59] Nilsson, S., Karlsson, G., "IP-address lookup using LC-tries", IEEE Journal on Selected Areas in Communications, Vol. 17, No. 6, 1999, pages 1083–1092, available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=772439

[60] Srinivasan, V., Varghese, G., "Faster IP lookups using controlled prefix expansion", in SIGMETRICS '98/PERFORMANCE '98. ACM, 1998, pages 1–10, available from: http://doi.acm.org/10.1145/277851.277863

[61] Tzi-cker Chiueh, Pradhan, P., "High performance IP routing table lookup using CPU caching", in INFOCOM, 1999, pages 1421-1428.

[62] Song, H., Hao, F., Kodialam, M. S., Lakshman, T. V., "IPv6 lookups using distributed and load balanced bloom filters for 100 Gbps core router line cards", in INFOCOM, 2009, pages 2518-2526.

[63] Fu, J., Hagsand, O., Karlsson, G., "Performance evaluation and cache behavior of LC-trie for IP-address lookup", in IEEE Workshop on High Perf. Switching and Routing, Poznan, 2006.

[64] Uzmi, Z. A., Nebel, M., Tariq, A., Jawad, S., Chen, R., Shaikh, A., Wang, J., Francis, P., "SMALTA: practical and near-optimal FIB aggregation", in CoNEXT '11. ACM, 2011, pages 29:1–29:12, available from: http://doi.acm.org/10.1145/2079296.2079325

[65] Song, H., Turner, J., Lockwood, J., "Shape shifting tries for faster IP route lookup", in Proceedings of the 13TH IEEE International Conference on Network Protocols, ser. ICNP '05. Washington, DC, USA: IEEE Computer Society, 2005, pages 358–367, available from: http://dx.doi.org/10.1109/ICNP.2005.36

[66] Leu, S., Chang, R.-S., "A fast and scalable IPv4 and 6 address lookup algorithm", Computer Communications, Vol. 29, No. 16, 2006, pages 3020 - 3036, available from: http://www.sciencedirect.com/science/article/pii/S014036640500438X

[67] Bando, M., Chao, H. J., "FlashTrie: Hash-based prefix-compressed trie for IP route lookup beyond 100 Gbps", in Proceedings of the 29th conference on Information communications, ser. INFOCOM'10. Piscataway, NJ, USA: IEEE Press, 2010, pages 821–829, available from: http://dl.acm.org/citation.cfm?id=1833515.1833653

[68] Hasan, J., Vijaykumar, T. N., "Dynamic pipelining: making IP-lookup truly scalable", SIGCOMM Comput. Commun. Rev., Vol. 35, No. 4, Aug. 2005, pages 205–216, available from: http://doi.acm.org/10.1145/1090191.1080116

[69] Jiang, W., Wang, Q., Prasanna, V. K., "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup.", in INFOCOM. IEEE, 2008, pages 1786-1794, available from: http://dx.doi.org/10.1109/INFOCOM.2008.241

[70] Kumar, S., Becchi, M., Crowley, P., Turner, J., "CAMP: fast and efficient IP lookup architecture", in Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, ser. ANCS '06. New York, NY, USA: ACM, 2006, pages 51–60, available from: http://doi.acm.org/10.1145/1185347.1185355

[71] McAuley, A. J., Francis, P., "Fast routing table lookup using CAMs", in INFOCOM'93. Proceedings. Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, IEEE. IEEE, 1993, pages 1382–1391.

[72] Scudder, J., "Router scaling trends", in RIPE-54 Meeting, 2007.

[73] Rétvári, G., Tapolcai, J., Kőrös i, A., Majdán, A., Heszberger, Z., "Compressing IP forwarding tables: towards entropy bounds and beyond", in ACM SIGCOMM Computer Communication Review, Vol. 43, No. 4. ACM, 2013, pages 111–122.

[74] Rottenstreich, O., Tapolcai, J., "Optimal rule caching and lossy compression for longest prefix matching", IEEE/ACM Transactions on Networking (TON), Vol. 25, No. 2, 2017, pages 864–878.

[75] Yang, T., Xie, G., Liu, A. X., Fu, Q., Li, Y., Li, X., Mathy, L., "Constant ip lookup with fib explosion", IEEE/ACM Trans. Netw., Vol. 26, No. 4, Aug. 2018, pages 1821–1836, available from: https://doi.org/10.1109/TNET.2018.2853575

Acronyms

API application programming interface. 52, 53

ASIC application-specific integrated circuits. 1–3, 6–8, 60, 63

ATM Asynchronous Transfer Mode. 59

BGP Border Gateway Protocol. 3, 4, 7, 18–20, 23, 41, 52, 53, 58, 59, 62

CIDR Classless Interdomain Routing. 3, 59

CPU central processing unit. 2, 4–14, 16–18, 23, 26, 27, 29–32, 35, 36, 38, 43, 44, 46, 55–60, 62, 63

DDR double data rate. 11, 59

DMA direct memory access. 14

DPDK Data Plane Development Kit. 16

DRAM dynamic random access memory. 10–13, 16, 37, 60

DXR Direct-eXtend-Range. 4, 5, 16, 17, 19, 22, 23, 26–39, 41–43, 46, 52–56, 59–63

EQIX Equinix Internet Exchange. 27

FIB forwarding information base. 27, 31, 36, 41, 44, 45, 52, 54, 58, 60, 63

FPGA field-programmable gate array. 3, 59, 60, 63

Gbps gigabits per second. 15, 16, 63

Glps billion routing lookups per second. 62

GP general-purpose. 2

GPU graphical processing unit. 2

HWPMC hardware performance monitoring counters. 35, 37

I/O input / output. 54

IC integrated circuit. 3

ILP instruction-level parallelism. 8, 9, 13, 62

IPC instructions per cycle. 9

kpps thousand packets per second. 15

LINX London Internet Exchange. 31–34, 37

LPM longest prefix matching. 3, 4, 13, 15–17, 26, 41, 46, 52–56, 60, 62, 63

Mbps megabits per second. 15

Mlps million routing lookups per second. 6, 27, 36, 46, 57, 58, 60, 62, 63

Mpps million packets per second. 2, 15, 53, 55, 56, 63

NIC network interface card. 14, 55

NUMA non-uniform memory access. 10, 13, 62

OoO out-of-order instruction scheduling and execution. 9, 27, 30, 54, 60

OS operating system. 2, 4, 15, 52

PATRICIA Practical Algorithm to Retrieve Information Coded in Alphanumeric. 15, 26

QDRII quad data rate II. 58

SDN software defined networking. 3

SDRAM synchronous dynamic random access memory. 59

SMP symmetric multiprocessing. 10

SMT simultaneous multi-threading. 12, 14, 38

SRAM static random access memory. 58

TCAM ternary content-addressable memory. 59, 60

TLB translation lookaside buffer. 13

VM virtual memory. 13

XORP eXtensible Open Router Platform. 15

Curriculum Vitae

Marko Zec was born in 1971 in Zagreb. He received the Dipl. Ing. degree in Electrical Engineering from the University of Zagreb in 1997. From 1996 until 2005 he worked as a systems and network administrator, designer and consultant, at the Ruđer Bošković Institute, IBM, AT&T, and local system integration companies, when his assigments included design, deployment, and management of nation-wide and campus networks for major government agencies. In 2005 he joined the Department of Telecommunications of the University of Zagreb, Faculty of Electrical Engineering and Computing (FER) where he currently holds the position of an associate researcher.

The main areas of his interest are computer networks, operating systems and programmable logic. His pioneering work from 2002 at virtualizing networking state in a general-purpose operating system was further developed at FER and merged into the mainline FreeBSD kernel in 2008, while the concept was later embraced by Linux and Solaris as well. At that time novel, network stack virtualization technology became the foundation for a popular network emulation tool called IMUNES, which together with professor Miljenko Mikuc he developed with initial funding from Croatian Ministry of Science (2004-2005). In 2004 he helped establishing research collaboration between FER and the International Computer Science Institute (ICSI), University of California, Berkeley. His major subsequent projects were backed by international and industrial funding: XORP (ICSI Berkeley, 2004-2006); VIRTNET (FreeBSD Foundation, 2007-2008); NXIX (Boeing Integrated Defense Systems, 2008-2012); E-IMUNES (Ericsson Nikola Tesla, 2012-2016).

In addition to IMUNES, which became a standard teaching tool used in several computer networks related courses at FER as well as at numerous universities worldwide, his educational contributions comprise the introduction of practical, FPGA-based laboratory exercises accompanying a digital logic course. This included design of low-cost FPGA development boards and IP components ranging from simple sequential logic blocks to complex modules such as a pipelined CPU core, multi-ported RAM controllers, and video frame buffers.

Marko Zec published eight peer-reviewed papers in international journals and conferences, delivered seven invited talks, and has appered as a speaker at diverse international technical conventions.

He lives in Zagreb with his wife and their three kids.

List of publications

Journals

- Zec, M., Rizzo, L., Mikuc, M. DXR: towards a billion routing lookups per second in software. ACM Computer Communications Review, Vol 42 Issue 5, October 2012.

Conference proceedings

- Zec, M., Mikuc, M. Pushing the Envelope: Beyond Two Billion IP Routing Lookups per Second on Commodity CPUs. (Best paper award). SoftCOM, Split, 2017.
- Salopek, D., Vasić, V., Zec, M., Mikuc, M., Vašarević, M., Končar, V. A network testbed for commercial telecommunications product testing. SoftCOM, Split, 2014.
- Zec, M., Mikuc, M. Operating system support for network emulation in IMUNES. OASIS / ASPLOS XI, Boston, MA, 2004.
- Zec, M. Implementing a clonable network stack in the FreeBSD kernel. USENIX Annual Technical Conference, FREENIX Track. San Antonio, Texas, 2003.
- Zec, M., Mikuc, M. Real-Time IP Network simulation at gigabit data rates. ConTEL, Zagreb, 2003.
- Zec, M., Mikuc, M., Žagar, M. Estimating the impact of interrupt coalescing delays on steady state TCP throughput. SoftCOM, Split, 2002.
- Musa, N., Zec, M., Kos, M. A method for managing distributed IP packet forwarding in ATM/LANE based networks. MIPRO, Rijeka, 2002.

Invited talks

- Area / speed tradeoffs in a retargetable FPGA-optimized processor core. Center for Embedded and Cyber-Physical Systems, University of California, Irvine, 15. 07. 2016.
- Operating system kernel as a building block for scalable and fast network topology emulation. Forschungzentrum Telekomunikation Wien, 18. 05. 2009.
- IMUNES: project status and future goals. Siemens AG, Munich, 11. 05. 2009.
- Operating system kernel as a building block for scalable and fast network topology emulation. Center for Embedded Computer Systems, University of California, Irvine, 25. 05. 2007.
- Network stack virtualization in FreeBSD kernel. Information Sciences Institute, University of Southern California, Marina del Rey, 20. 10. 2004.
- Using an operating system kernel as a building block for scalable and fast network topology emulation. International Computer Science Institute, University of California, Berkeley, 27. 10. 2004.

- Clonable / virtualized BSD network stack: implementation, performance, applications. Apple Inc., Cupertino, 20. 05. 2004.

International conference and workshop appearances

- Zec, M., Jadrijević, D. FPGArduino: A Cross-Platform RISC-V IDE for masses. 4th RISC-V workshop, MIT, Cambridge, MA, 12. 07. 2016.

- Zec, M. Network emulation using the virtualized network stack in FreeBSD. MeetBSD 2010, Krakow, Poland, 03. 07. 2010.

- Zec, M. Network stack virtualization for FreeBSD 7.0. BSDCan 2007, Ottawa, 18. 05. 2007.

- Zec, M. Towards and beyond network stack virtualization in the FreeBSD kernel. NLUUG 2007, Ede, The Netherlands, 10. 05. 2007.

- Zec, M. FreeBSD network stack virtualization. EuroBSDCon, Amsterdam, 16. 11. 2002.

Životopis

Marko Zec rođen je 1971. godine u Zagrebu. Diplomirao je 1997. na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Od 1996. do 2005. bio je zaposlen kao administrator operacijskih sustava i računalnih mreža, te kao projektant i konzultant, na Institutu Ruđer Bošković, u IBM Hrvatska, AT&T Hrvatska, te drugim tvrtkama za integraciju IT sustava. Tijekom tog razdoblja njegove projektne zadaće obuhvaćale su projektiranje, implementaciju, te nadzor i upravljanje nekoliko velikih WAN i campus mreža tijela državne uprave.

2005. godine zapošljava se na Zavodu za telekomunikacije Fakulteta elektrotehnike i računarstva kao zavodski suradnik na istraživačkim projektima, gdje 2013. godine prelazi na mjesto višeg laboranta, a od 2017. je na radnom mjestu stručnog suradnika u Zavodu.

Glavna područja interesa su mu računalne mreže, operacijski sustavi, te programirljiva logika. Njegov rad iz 2002. na virtualizaciji mrežnog stoga operacijskog sustava opće namjene integriran je u standardnu jezgru operacijskog sustava FreeBSD, a isti je koncept kasnije prihvaćen i u operacijskim sustavima Solaris i Linux. Koncept emulacije topologije računalnih mreža korištenjem virtualiziranog mrežnoG stoga postao je temelj alata IMUNES, kojeg je zajedno s prof. Mikucem započeo razvijati uz potporu MZOS (2004/2005). Kao vanjski suradnik FER-a, 2004. godine pomaže pri uspostavi istraživačke suradnje između FER/a i International Computer Science Institute, University of Califoria, Berkeley. Projekti na kojima je kasnije radio bili su financirani kroz međunarodnu suradnju i / ili od strane gospodarstva: XORP (ICSI Berkeley, 2004-2006); VIRTNET (The FreeBSD Foundation, 2007/2008); NXIX (Boeing Integrated Defense Systems, 2008-2012); E-IMUNES (Ericsson Nikola Tesla, 2012-2016).

Uz IMUNES koji je usvojen kao standardni alat na nekoliko kolegija iz područja računalnih mreča na FER-u te na nizu drugih sveučilišta širom svijeta, doprinio je unaprijeđenju nastave i kroz razvoj praktičnih laboratorijskih vježbi iz digitalne logike, što uključuje i razvoj cijenom dostupnih FPGA razvojnih pločica, te širokog skupa logičkih modula, od jednostavnih sekvencijskih blokova do procesorske jezgre, vanjskih RAM sučelja s integriranim arbitrom, generatora slike itd.

Objavio je osam radova s međunarodnom recenzijom u časopisu i na konferencijama te je održao niz predavanja na međunarodnim stručnim skupovima. Pozvana predavanja održao je u Apple Inc., Cupertino; ICSI, UC Berkeley; ISI, University of Southern California; UC Irvine;

Siemens AG, München; i Forschungzentrum Telekomunikation, Beč.

Oženjen je i otac troje djece.