BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

HIGH-PERFORMANCE SOFTWARE PACKET PROCESSING

by

QIAOBIN FU

B.E., Dalian University of Technology, 2011
M.S., University of Chinese Academy of Sciences, 2014

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2020

Approved by

First Reader

John W. Byers, Ph.D.
Professor of Computer Science

Second Reader

Mark Crovella, Ph.D.
Professor of Computer Science

Third Reader

Azer Bestavros, Ph.D.
Warren Distinguished Professor of Computer Science

Fourth Reader

Abdul Kabbani, Ph.D.
Senior Engineer
Google

The number of transistors on a microchip doubles about every two years, though the cost of computers is halved.

Gordon Moore (1965)

Moore's Law can't continue forever … We have another 10 to 20 years before we reach a fundamental limit.

Gordon Moore (2005)

Acknowledgments

First and foremost, I would like to thank my advisor Prof. John W. Byers, who has given me tremendous support during my Ph.D. journey. Besides, John always gave me the flexibility to pursue my professional career and guided me to explore new opportunities.

I also would like to thank my XIA teammates, Michel Machado and Cody Doucette. As a good team, we did a lot of knowledge sharing and discussion, and everyone has a passion for perfection. As Michel said: “we are exploring the edge of possibility”. Our project wouldn’t be as successful as it is without each other’s help.

I thank my committee members and Michel Machado for their valuable suggestions.

I thank Prof. Azer Bestavros for his valuable advice on promising research directions. His trust has kept me moving forward even during my lowest moments. I am honored to be the Teaching Fellow five times for his signature course - CS350, from which I also learned a lot and saw his passion for education.

I thank my mentors during internships (in time order): Dr. Kate Keahey (Argonne National Laboratory/UChicago), Dr. Pierre Riteau (Argonne National Laboratory/StackHPC), Prof. Keon Jang (Google/MPI-SWS), Dr. Abdul Kabbani (Google), Dr. Nandita Dukkipati (Google), and Dr. Haohan Zhu (Facebook). Thank you all for mentoring me in new directions, I gained a lot of new skills.

I thank my lab-mates and friends, I really enjoyed the time with you all at BU. Fortunately, I got to know my best friend - Baichuan Zhou during the orientation at BU.

I also thank my collaborator and friend, Prof. Tong Yang (PKU), together we explored several interesting topics in high-speed networking during my Ph.D. journey. I thank my collaborators, Prof. Gaogang Xie (CAS) and Prof. Alex X. Liu (MSU).

Finally, I thank my family for their generous support. Especially, I would like to thank my wife, Tingting Xu, for her accompanying and support. Together we pursed our PhDs and had a wonderful time at BU.

HIGH-PERFORMANCE SOFTWARE PACKET PROCESSING

QIAOBIN FU

Boston University, Graduate School of Arts and Sciences, 2020

Major Professor: John W. Byers, PhD
Professor of Computer Science

ABSTRACT

In today's Internet, it is highly desirable to have fast and scalable software packet processing solutions for network applications that run on commodity hardware. The advent of cloud computing drives the continued rapid growth of Internet traffic. Moreover, the development of emerging networking techniques, such as Network Function Virtualization, significantly shapes the need for implementing the network functions in software. Finally, with the advancement of modern platforms as well as software frameworks for packet processing, network applications have potential to process 100+ Gbps network traffic on a single commodity server. Representative frameworks include the Click modular router, the RouteBricks scalable routing architecture, and BUFFALO, the software-based Ethernet switch. Beneath this general-purpose routing and switching functionality lie a broad set of network applications, many of which are handled with custom methods to provide cost-effectiveness and flexibility. This thesis considers two long-standing networking applications, IP lookup and distributed denial-of-service (DDoS) mitigation, and proposes efficient software-based methods drawing from this new perspective.

In this thesis, we first introduce several optimization techniques to accelerate network applications by taking advantage of modern CPU features. Then, we explore the IP lookup problem to find the longest matching prefix of an IP address in a set of prefixes. An ideal

IP lookup algorithm should achieve small constant IP lookup time, and on-chip memory usage. However, no prior IP lookup algorithm achieves both requirements at the same time. We propose SAIL, a splitting approach to IP lookup, and a suite of algorithms for IP lookup based on SAIL framework. We conducted extensive experiments to evaluate our algorithms, and experimental results show that our SAIL algorithms are much faster than well-known IP lookup algorithms. Next, we switch our focus to DDoS, an attempt to disrupt the legitimate traffic of a victim by sending a flood of Internet traffic from different sources. Our solution is Gatekeeper, the first open-source and deployable DDoS mitigation system. We present a series of optimization techniques, including use of modern platforms, group prefetching, coroutines, and hashing, to accelerate Gatekeeper. Experimental results show that these optimization techniques significantly improve its performance over alternative baseline solutions.

Contents

List of Tables

List of Figures

List of Abbreviations

APU	. . . .	Accelerated Processing Unit
AS	. . . .	Autonomous System
BGP	. . . .	Border Gateway Protocol
CIDR	. . . .	Classless Inter-domain Routing
CPU	. . . .	Central Processing Unit
DDoS	. . . .	Distributed Denial-of-Service
DPDK	. . . .	Data Plane Development Kit
DRAM	. . . .	Dynamic Random-Access Memory
eBPF	. . . .	extended Berkeley Packet Filter
ECMP	. . . .	Equal-cost multi-path routing
FIB	. . . .	Fowarding Information Base
FPGA	. . . .	Field-Programmable Gate Array
Gbps	. . . .	Giga-bits per second
GP	. . . .	Group Prefetching
GPU	. . . .	Graphics Processing Unit
GRO	. . . .	Generic Receive Offload
ICMP	. . . .	Internet Control Message Protocol
ISP	. . . .	Internet Service Provider
IXP	. . . .	Internet eXchange Point
JIT	. . . .	Just-in-time
L1D	. . . .	First Level Data Cache
LFB	. . . .	Line Fill Buffer
LLC	. . . .	Last Level Cache
LPM	. . . .	Longest Prefix Matching
LSB	. . . .	Least Significant Bit
MAC	. . . .	Media Access Control
Mpps	. . . .	Million packets per second
NFV	. . . .	Network Function Virtualization
NIC	. . . .	Network Interface Card
NUMA	. . . .	Non-Uniform Memory Access
OVS	. . . .	Open vSwitch
RDMA	. . . .	Remote Direct Memory Access
RED	. . . .	Random Early Detection
RSS	. . . .	Receive-Side Scaling

SAIL	....	Splitting Approach to IP Lookup
SDN	....	Software Defined Networking
SIMD	....	Single Instruction Multiple Data
SoC	....	System-on-Chip
SOL	....	Solicitor
TbE	....	Terabit Ethernet
TCAM	....	Ternary Content-Addressable Memory
TLB	....	Translation Lookaside Buffer
VP	....	Vantage Point
XDP	....	eXpress Data Path

Chapter 1

Introduction

1.1 Motivation

With the advent of cloud computing, we have seen and continue to see the proliferation of massive amounts of data, which can drive Internet traffic to grow rapidly. This brings significant challenges to the design of switches, routers, and middleboxes (*i.e.*, proprietary, and dedicated hardware appliances), as they are expected to process network traffic at the scale of multiple hundreds of gigabits per second (Gbps). Moreover, more and more applications demand ultra-low latency and high bandwidth, the link speed on commodity servers has grown from 1Gbps to 100Gbps over the past years and this growing trend is still continuing. The Terabit Ethernet (TbE) links are expected to be deployed in production soon [Jain, 2016]. Thus, there is a strong need for fast and scalable packet processing solutions for the switches, routers, middleboxes and commodity servers.

On the other hand, several emerging techniques heavily shape the need for *software packet processing* solutions that only run on general-purpose computers. For example, software-defined networking (SDN) [McKeown et al., 2008] is a new network design that separates the network control plane from the forwarding plane, and it provides flexibility and controllability to both network and data center operators. Complementary to SDN, network function virtualization (NFV) [Guerzoni et al., 2012] shifts the middlebox processing (*e.g.*, Intrusion Detection Systems, network address translation, load balancers and message routers, etc.) from proprietary hardware appliances owned by the network operators to software that can run on inexpensive and commodity hardware. Thus it reduces cost and enables fast and elastic resource provisioning. In the era of cloud computing, virtualization is the key enabler for these emerging techniques: it supports multi-tenancy and combined with the usage of virtual switches (*e.g.*, Open vSwitch [Pfaff et al., 2015]), it allows enforcement of complex routing policies, (*e.g.*, packets belonging to different flows are processed by different network functions) from the SDN controller over the packets. Many cloud providers use these technologies to power their cloud platforms: NVP [Koponen et al., 2014] is an SDN-based network virtualization platform that has been deployed in many production environments. The Google Cloud Platform uses Andromeda [Dalton et al., 2018], Microsoft Azure uses VFP [Firestone, 2017], and VMware uses VDS [VMware, 2012] for network virtualization. Therefore, it is highly desirable to have high-performance packet processing in software.

1.2 Network Applications and Challenges

As mentioned in the above section, driven by the development of NFV, the network operators and cloud providers are replacing hardware appliances with virtualized network functions to allow networks to be agile and capable of responding to the needs of new services and traffic. Especially, several frameworks have been proposed to deal with the challenges on the design of switches, and routers. Representative frameworks include the Click modular router [Kohler et al., 2000], the RouteBricks scalable routing architecture [Dobrescu et al., 2009], the GPU-accelerated software router - PacketShader [Han et al., 2010], as well as BUFFALO [Yu et al., 2009] and CuckooSwitch [Zhou et al., 2013], the software-based Ethernet switches. Beneath this general-purpose routing and switching functionality lie a broad set of network applications, many of which are handled with custom methods to provide cost-effectiveness and flexibility. Besides, there is wide body of work [Sekar et al., 2012, Yang et al., 2014, Martins et al., 2014, Sherry et al., 2015, Yang et al., 2016, Yang et al., 2018a, Liu et al., 2019c] on improving the performance of network applications, such as load balancer, network measurement, IP lookup, virtual switches, etc. It is increasingly important to achieve high performance for these fundamental networking functionalities.

Among these applications, the problem of fast IP lookup has long been a core networking issue and the bottleneck of software routers. It has broad applications, as it runs on every router and is the building block for many network applications as well, such as firewalls, and network address translation. Another long-standing problem is the DDoS attack, an attempt to disrupt the legitimate traffic of a victim by sending a flood of Internet traffic from different sources. DDoS mitigation solutions are needed to protect the victims from these attacks. Gatekeeper [Machado et al., 2020] is the first open-source and deployable DDoS mitigation system that we have been building for the past five years. Gatekeeper aims to achieve 10 Gbps throughput on a single commodity server for the first deployment, and its ultimate goal is to achieve 100+ Gbps throughput. These two applications (*i.e.*, IP lookup, and DDoS mitigation) that need high-performance software packet processing are the main focus of this thesis, which are heavily optimized by taking advantage of the advancement in modern platforms and network I/O as discussed in the following section.

1.3 Modern Platforms and Network I/O

The server landscape has changed a lot over the past few years: modern servers are typically equipped with advanced hardware resources (*e.g.*, multi-core CPUs, NUMA support, SmartNICs, faster DRAM and PCIe, etc.) to achieve high performance. The advancement of these technologies makes it feasible to achieve high-performance software packet processing on commodity servers. In the following, we highlight three technologies that are quite relevant to this thesis:

First, the performance of CPUs has increased significantly over the past decade, thanks to the major improvement in better parallelism (*e.g.*, a larger number of cores), larger cache sizes, larger memory bandwidth, etc. Typically, modern CPUs have advanced technologies (*e.g.*, Intel QuickData DMA Engines [Intel, 2005], Intel Data Direct I/O Technology [Intel, 2012], Accelerated Processing Units [Go et al., 2017]) to optimize the bottleneck I/O of the applications, which can significantly improve their performance.

Second, modern network interface cards (NICs) [Li et al., 2015, Intel, 2019b] have driven rapid improvement in bandwidth and latency, and they offer several advanced features (*e.g.*, multi-queue, receiver-side scaling, and flow-steering) that allow to fully utilize the potentials of the modern multi-/manycore CPUs. More specifically, the multi-queue support allows different CPU cores to access the NIC without contending with each other. With receiver-side scaling (RSS) support, the NIC can distribute a subset of incoming packets to different CPU cores with an in-order delivery guarantee based on flow hash values. With flow steering, NICs can steer ingress traffic to dedicated receive queues according to the flow classification rules installed by the network applications. Moreover, with the end of Moore's law, the only path left to improve energy-performance-cost is specialization [Hennessy and Patterson, 2017]. SmartNICs are recently proposed to deal with the ever-growing network link speed at a lower cost. There is a wide body of ongoing research efforts [Li et al., 2017, Firestone et al., 2018, Liu et al., 2019a, Liu et al., 2019b, Choi et al., 2019, Gai, 2020] from both industry and academia. Typically, a SmartNIC integrates different hardware resources on a board (*e.g.*, CPU cores, DRAM, DMA engines, network accelerators for fixed functions, ASIC-based data plane, RDMA, etc.). With the advancement of the on-board hardware, SmartNICs will have more powerful CPUs, large caches (*e.g.*, around 30 MB LLC), large DRAM (more than 8 GB), etc. With enough hardware resources, not only the whole network stack can be offloaded to the NICs, but also part or even all of the application logic can be offloaded, as some data structures and algorithms can be efficiently implemented on these NICs. Overall, SmartNICs are promising solutions to the ever-growing high-speed networks.

Third, as the OS network stack cannot keep pace with the ever-growing fast network links, several software packet processing frameworks (*e.g.*, PacketShader I/O [Han et al., 2010], netmap [Rizzo, 2012], Intel DPDK [DPDK Project, 2010], Fastclick [Barbette et al., 2015], PF_RING [ntop, 2019]) have been proposed to avoid the heavy overheads of the OS network stacks and provide line-rate network I/O for very high-speed network links.

Therefore, by taking advantage of the modern platforms, network applications implemented on these frameworks even have the potential to deal with 100+ Gbps network traffic (148.8+ million packets per-second for minimum-size 64 bytes packets) on a single commodity server.

1.4 Summary of Contributions

In this thesis, we systematically summarize the acceleration techniques by taking advantage of the modern CPU features (*e.g.*, cache hierarchy, out-of-order execution, optimized network I/O) to improve the performance of software packet processing applications. Specifically, we summarize several optimization techniques, including compact data structure design for better cache hits, batching as well as memory prefetching. To the best of our knowledge, we are the first to introduce coroutines to the context of packet processing in network systems.

Then, we explore one of the fundamental networking functionalities that have broad applications: IP lookup. An ideal IP lookup algorithm should achieve small constant IP lookup time, and on-chip memory usage. However, no prior IP lookup algorithm achieves both requirements at the same time. We make three key contributions in this thesis. First, we propose SAIL, a splitting approach to IP lookup mainly for IPv4. One splitting is along the dimension of the lookup process, namely finding the prefix length and finding the next hop, and another splitting is along the dimension of prefix length, namely IP lookup on prefixes of length less than or equal to 24 and that longer than 24. Second, we propose a suite of algorithms for IP lookup based on our SAIL framework. Third, we extend our SAIL approach to IPv6. Since the IPv6 address is much longer than IPv4 address, SAIL would require much larger on-chip memory usage for IPv6 than that for IPv4 in the worst case. We proposed a hybrid scheme to address this issue. To improve the performance of synthetic large IPv6 FIBs in the worst case, we propose a novel technique called *pivot inheriting*. We conducted extensive experiments to evaluate our algorithms using real FIBs and real traffic from a major ISP in China. Experimental results show that our SAIL algorithms are much faster than well-known IP lookup algorithms.

Finally, we switch our focus to improving the performance of Gatekeeper, the first open-source and deployable DDoS mitigation system that we have been building for the past five years. Gatekeeper is implemented based on Intel's DPDK, a set of libraries for high-speed packet I/O. We present a series of optimization techniques, including batching, group prefetching and coroutines, to improve Gatekeeper's performance. Compared with the vanilla implementation of Gatekeeper, we improve on its performance by a factor of more than $90\times$. These optimizations make Gatekeeper ready for the first deployment with a single 10Gbps Gatekeeper server. As the ultimate goal of Gatekeeper is to achieve 100+ Gbps throughput on a single commodity server, and the flow hash table is on the critical path, this thesis further conducted experiments on various hash table algorithms to evaluate their performance. The results show that hopscotch hashing and separate chaining outperform the others, which can guide the redesign of the flow hash table in Gatekeeper.

The contributions of this dissertation also include the following:

Our SAIL algorithms are cross platform as the data structures are all arrays and only require four operations of ADD, SUBTRACTION, SHIFT, and logical AND. SAIL algorithms can be implemented on at least four platforms (namely FPGA, CPU, GPU, and many-core). Moreover, we implemented an open-source environment [Yang, 2014] for comparing the five well-known IP lookup algorithms (namely LC-trie, Tree Bitmap, Lulea, DXR, and SAIL_L).

An open-source, fully functional implementation of Gatekeeper [Machado et al., 2020] is available online for download. This work is in collaboration with Michel Machado and Cody Doucette. The implementation complies with industry practices and standards, and it is ready for production deployment. To the network community, Gatekeeper is a template to make other architectural solutions deployable by a single AS. Moreover, we explored a feasible roadmap to accelerate Gatekeeper to achieve 100+ Gbps throughput on a single commodity server, including acceleration using SmartNICs.

Over the past few years, we have made consistent efforts to improve the performance of software packet processing in network applications. To achieve the highest potential performance for software packet processing, we have to optimize the network applications in a holistic way. We will discuss the principles and lessons learned that generally guided the performance improvement of software packet processing during our work in §5.1.

1.5 Thesis Outline

We structured the chapters as follows: Chapter 1 motivates the need for high-performance software packet processing and introduces the problems that this thesis focuses on. Chapter 2 gives some background on high-performance software packet processing as well as the two applications that this thesis tries to accelerate. Moreover, it explores the state-of-the-art optimization techniques to accelerate software packet processing by taking advantage of modern CPU features. Finally, it discusses the future of NICs. Chapter 3 presents our splitting approach to IP lookup - SAIL, and compares it with the state-of-the-art algorithms. Chapter 4 introduces Gatekeeper, the first open-source and deployable DoS mitigation system. Moreover, it shows the optimization techniques used to improve the performance of Gatekeeper, so that Gatekeeper can meet the challenging requirements in production. Chapter 5 concludes our work.

Chapter 2

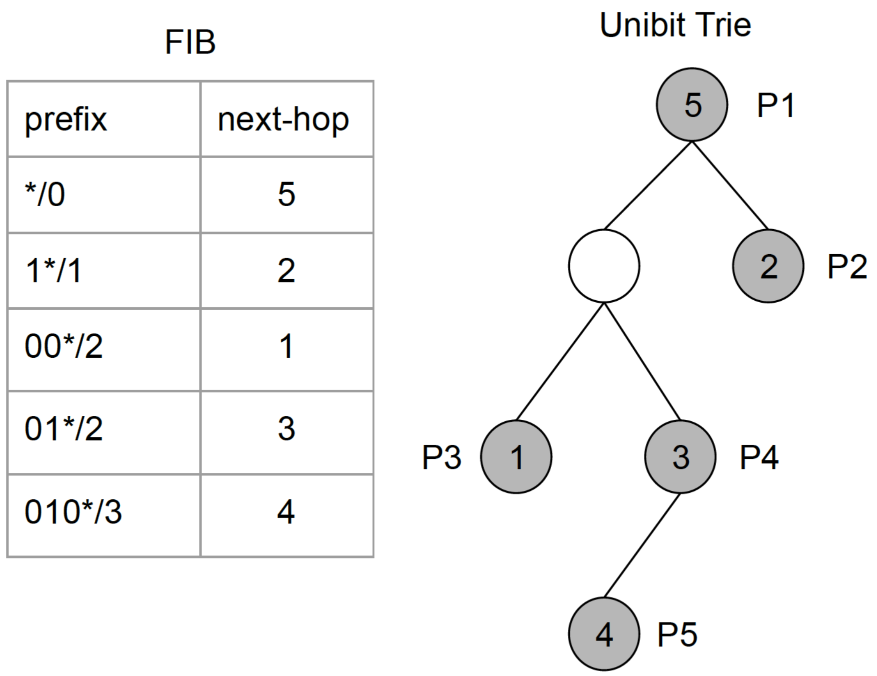
Background

This chapter first gives some background on the two problems that this thesis focuses on - (1) IP lookup problem; (2) Denial-of-Service (DoS) attack problem. Next, we briefly introduce Intel DPDK, which is a fast packet processing framework for userspace network I/O. Because we heavily rely on DPDK to build our open-source and deployable DDoS mitigation system as discussed in Chapter 4. Then, we switch our focus to software packet processing acceleration techniques, which are motivated by the fact that the performance gap between CPU and memory is increasing as we will discuss in §2.4. To address this performance issue, we take a close look at the CPU cache hierarchy as well as several hardware features available in modern CPUs, and introduce several data prefetching techniques to reduce memory stalls significantly, including hardware prefetching, group prefetching and coroutines-based prefetching. To the best of our knowledge, we are the first to bring coroutines-based prefetching to the context of software packet processing in network systems. Finally, we discuss the future of NICs to deal with the ever-growing network data. Specifically, with the end of Moore's law, CPUs cannot keep up with the ever-growing network data for network applications. We may have to offload some of the tasks that the host CPU would normally handle to specialized hardware. Therefore, we give an overview of common hardware choices, discuss their advantages and disadvantages, and then introduce recent research efforts on SmartNICs.

2.1 IP Lookup: Problem Definition

The Internet Protocol (IP) is one of the most important protocols in the Internet, and it provides logical communication between hosts. IP protocol is widely used: each interface on every host and router in the Internet must have a globally unique IP address to communicate with each other. Basically, IP establishes the Internet. There are two major verions of IP, Internet Protocol Version 4 (IPv4) and its successor - Internet Protocol Version 6 (IPv6). Each IPv4 address has 32 bits (*i.e.*, 4 bytes), and there are a total of 2^{32} (about 4 billion) possible IPv4 addresses. IPv4 addresses are typically written in a format called dotted-decimal notation, in which each byte of the address is written in its decimal form and is separated by a period (.) from other bytes in the address. Let us take the IPv4 address `41.76.232.0` for example. The `41` is the decimal equivalent of the first 8 bits of the IPv4 address, the `76` is the decimal equivalent of the second 8 bits of the IPv4 address, and so on. Thus, the address `41.76.232.0` in binary notation is `00101001 01001100 11101000 00000000`. Each IPv6 address has eight groups of four hexadecimal digits, and each group has 16 bits. The groups are separated by colons (:). Let us take the IPv6 address `2001:0ce5:66bc:7edf:0000:1230:0427:2020` for example. The `2001` is the hexadecimal equivalent of the first 16 bits of the IPv6 address, the `0ce5` is the hexadecimal equivalent of the second 16 bits of the IPv6 address, and so on. Thus, the address `2001:0ce5:66bc:7edf:0000:1230:0427:2020` in binary notation is `0010000000000001 0000110011100101 0110011010111100 0111111011011111 0000000000000000 0001001000110000 0000010000100111 0010000000100000`. To compactly represent an IP prefix, the Classless Inter-domain Routing (CIDR) notation [Fuller et al., 1993] is typically used. A CIDR notation consists of three parts: an IP address, a slash (/) character, and an integer. The integer represents the prefix length (*i.e.*, the number of leading 1 bits in the subnet mask).

When an incoming packet arrives at a router on an IP-addressed internetwork, the router uses the packet's destination IP address to look up the output link interface in its forwarding table, which maps destination addresses to link interfaces. After identifying the output link interface, the router forwards the packet to that output link interface. Note that, there are possibly multiple matches in the router's forwarding table for a given IP address, the well-known *longest prefix matching (LPM)* rule is used to find the longest prefix in the table and forward the packet to the link interface associated with the longest prefix match.



FIB

prefix	next-hop
*/0	5
1*/1	2
00*/2	1
01*/2	3
010*/3	4

Figure 2·1: IP lookup example with unibit trie.

We now describe how IP lookup works using one fundamental data structure and algorithm for the longest prefix matching - unibit trie. Most of the well-known algorithms are variants of the unibit trie. Given a FIB table, we first construct a unibit trie. An example trie is in Figure 2·1. CIDR notation is used to represent the prefix. Based on whether a node represents a prefix in the FIB, there are two types of nodes: *solid nodes* and *empty nodes*. A node is solid if and only if the prefix represented by the node is in the FIB. That is, a node is solid if and only if it has a next hop. A node is an empty node if and only if it has no next hop. Each solid node has a label denoting the next hop of the prefix represented by the node. Give an IP address, the IP lookup process starts at the root node of the trie. Based on the value of each bit of the IP address, the lookup procedure decides to visit either the left (when value is 0) or the right node (when value is 1). The latest solid node found along the path is maintained as the best matched prefix node while the trie is traversed. Let us take the IP address `0101` for example. The IP lookup starts at the root node (*i.e.*, node P1), traverses the path indicated by the IP address, and remembers the best matched prefix node so far. The first bit of `0101` is 0, so it goes to the left node of node P1 and gets to the node 0*, which is an empty node. The second bit of `0101` is 1, so it goes to the right node and gets to the node 01* (*i.e.*, node P4), which is a solid node. The algorithm remembers node P4 as the best matched prefix node so far. The third bit of the IP address is 0, and it goes to the left node (*i.e.*, node P5), which is a leaf node. It is a solid node, so the best matched prefix node becomes node P5. Finally, its associated next hop information (*i.e.*, 4) is returned and the IP lookup terminates.

2.2 DoS Attack Problem

In a DoS attack, an adversary attempts to disrupt the legitimate traffic of a targeted server, service, or network by overwhelming the target or its surrounding infrastructure with a flood of Internet traffic. Typically, there are a few (or even a single) attackers sending attack traffic, for example in IP spoofing [CERT Coordination Center, 1996]. With the evolution of DoS attacks, the distributed DoS (DDoS) attacks became the norm. Specifically, today's attacker typically has remote control of thousands of online hosts (*e.g.*, computers, IoT devices) infected with malware, which is called a botnet. When the attacker wants to attack a victim, for example, specified by an IP address, the attacker sends attack command to each bot in the botnet, and the bots will send requests to the target, overwhelming the host, potentially resulting in a denial-of-service to legitimate users.

DoS attacks are growing exponentially in terms of both capacity and frequency over the past 15 years. With a recent attack exceeding 1.7Tbps in size [NETSCOUT, 2019], DoS attacks could have a devastating impact on the victims. Moreover, it is prohibitively expensive to mitigate large DoS attacks while waging an attack is cheap and simple, making DoS attacks the top concern to many Internet actors. By the numbers, more than 23% of the entities surveyed by Arbor Networks indicated the cost of a major DoS attack at above \$100K, and 5% cited the cost at over \$1M [Anstee et al., 2017, Figure 32]. In contrast, attackers can easily launch a 125Gbps DoS attack for only several dollars [Makrushin, 2017]. As suggested, the average cost of DoS attacks from 2018 to 2023 will only fall since the attack surface and resources leveraged by attackers are growing fast: the average broadband speed is more than doubling, and the number of networked devices is increasing by around 60% [Cisco, 2020].

2.3 Intel DPDK

The Linux network stack is too slow to keep up with very high-speed network traffic (*i.e.*, 100+ Gbps/node) [Li et al., 2015]. Several software frameworks have been proposed to allow the network applications to avoid the heavy overheads of the OS network stacks and provide line-rate network I/O for very high-speed links, even with minimum-sized packets. Three representative frameworks are discussed as follows: (1) PacketShader framework [Han et al., 2010] can accelerate the general packet processing with Graphics Processing Unit (GPU). On one hand, they proposed several techniques to optimize PacketShader's packet I/O engine, including batch processing, use of per-queue counters instead of per-NIC counters to eliminate CPU contention, use of NUMA-aware data placement and I/O to minimize packet movement between local and remote memory. On the other hand, PacketShader uses GPUs to offload computation and memory-intensive workloads, taking advantage of the massively-parallel processing capability of GPU. (2) Netmap [Rizzo, 2012] is a framework that enables commodity operating systems to handle $1-10$Gbps links without requiring custom hardware or changes to applications. Three key techniques are used to achieve this goal: (a) removing the need for per-packet dynamic memory allocations by preallocating resources; (b) reducing the system call overheads by using large batches; (c) eliminating memory copies by carefully sharing buffers and metadata between kernel and userspace. (3) PFRING ZC (Zero Copy) framework [ntop, 2019] achieves 10Gbps packet processing at any packet size by using zero copy operations similar to netmap. However, PFRING ZC supports huge pages and per-NUMA node buffer regions, but users will need to pay for the licenses. Among these software frameworks, the Intel Data Plane Development Kit (DPDK) community involves industry efforts to enable fast packet processing. Specifically, DPDK is a set of libraries to accelerate packet processing workloads running on a wide variety of CPU architectures.

2.3.1 DPDK Architecture

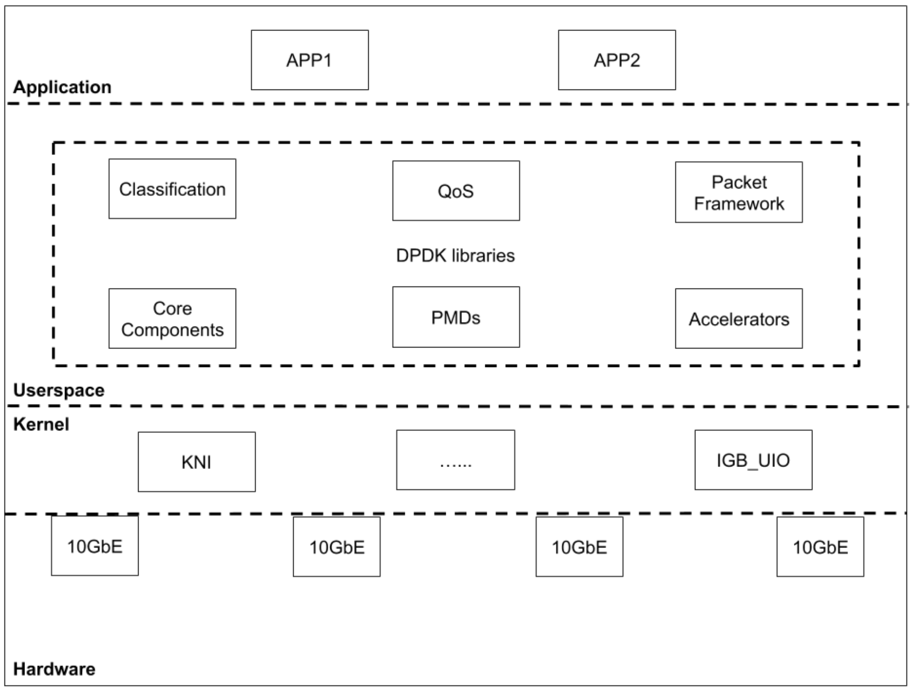


Figure 2·2: DPDK architecture.

Figure 2·2 presents DPDK architecture. DPDK has several kernel modules that allow DPDK applications to interact with the kernel. For example, the DPDK Kernel NIC Interface (KNI) allows DPDK applications access to the Linux control plane; IGB UIO allows DPDK applications to deal with PCI enumeration and handle links status interrupts, instead of being handled by the kernel. Moreover, DPDK provides a set of libraries to accelerate packet processing workloads, such as Poll Mode Drivers (PMDs), Classification library, QoS library, etc. Specifically, DPDK allows applications built on top of it to bypass the kernel by placing device drivers in user-space, and is optimized to receive and send packets within the minimum number of CPU cycles. As DPDK states: "some packet processing functions have been benchmarked up to hundreds of million frames per second, using 64-byte packets with a PCIe NIC" [DPDK Project, 2010]. All user-space drivers operate in polling mode, eliminating interrupt overhead. All PMDs in DPDK support two packet processing models: (1) *run-to-completion model* allows applications to handle both the packet I/O and application workload on a single core. The packet I/O can be scaled over multiple cores; (2) *pipeline model* allows I/O applications to disperse packets to other cores and other cores will perform the application work. In this mode, the I/O application can handle more I/O on fewer cores with vectorization. The core components in DPDK provide all the elements needed for high-performance packet processing applications, like memory management. Recently, DPDK provides a set of libraries for accelerators (*e.g.*, ARM, FPGA, etc.), which allow the developers to adopt the SmartNIC-based solutions in the near future. Among these libraries, DPDK applications heavily rely on the following features to achieve high performance: memory management and burst packet I/O.

2.3.2 Memory Management

DPDK provides NUMA (Non-Uniform Memory Access)-aware memory allocation as well as hugepages support [DPDK Project, 2010], which are critical for DPDK applications to achieve high performance.

NUMA

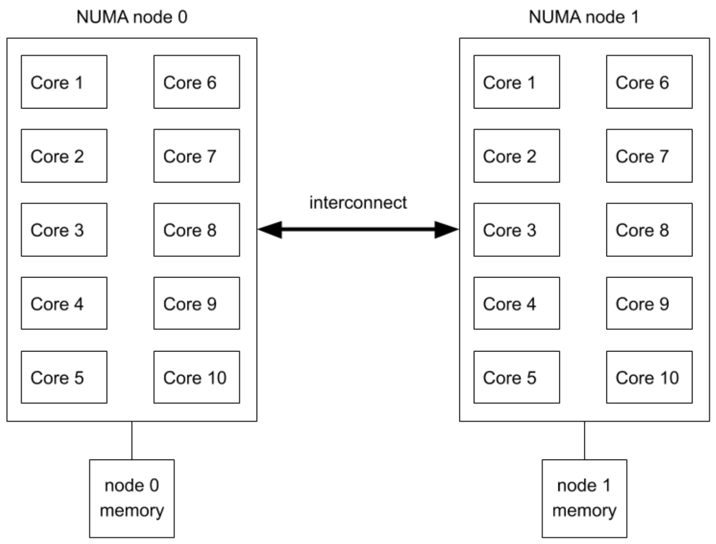


Figure 2·3: A system with two NUMA nodes and twenty CPU cores.

Today, NUMA becomes common due to two main reasons: (1) the processors' performance are increasing, and the memory should be directly attached to the socket that the processors are on to minimize the memory access latency [Lameter, 2013]; (2) In a multiprocessor architecture, the bandwidth demands become larger with a larger number of processors. The memory system in such architectures is typically distributed among the processors, otherwise, it will incur excessively long access latency [Hennessy and Patterson, 2017]. Figure 2·3 shows an example NUMA system with two NUMA nodes. Each node belongs to each socket, with ten CPU cores each. The two CPUs are connected with each other via interconnection links. The Intel Quickpath Interconnect (QPI) links [Intel, 2009] are used by Intel Xeon family CPUs.

In NUMA systems with multiple CPUs, memory access time depends on the physical memory location relative to the processor. Processors can directly access their local memory with low latency. However, remote memory access requires an extra hop via the hosting CPU node, thus remote memory accesses typically take much longer time than local memory accesses. In [Han et al., 2010], experimental results show that remote memory access in a NUMA system increased the access time by 40-50% and lowered the bandwidth by 20-30% compared to local memory access. Therefore, remote memory accesses in a NUMA system can lead to significant performance degradation, and they should be avoided as much as possible.

HugePages

In modern CPUs, memory is managed by using pages, which are contiguous blocks of memory. Applications use page tables to translate virtual addresses used by the software applications to physical addresses used by the hardware. The page tables map virtual to physical addresses, on a page level of granularity. On Intel CPUs, the standard system page size is 4 KB. Caches use the services of the Instruction Translation Lookaside Buffer (ITLB), Data Translation Lookaside Buffer (DTLB) and Shared Translation Lookaside Buffer (STLB) to translate linear addresses to physical addresses. However, the TLBs are pretty small in size, so that the amount of memory covered by the TLBs for the standard page sizes is very small (only a few megabytes). Therefore, applications with large memory requirements can suffer from a high rate of TLB misses. To address this performance issue, modern CPUs have HugePage support. Currently, Intel CPUs have two available HugePage sizes: 2 MB and 1 GB [Intel, 2019a]. Therefore, a single page can cover an entire 2 MB or 1 GB physically and virtually contiguous memory area, and the TLBs can cover up to several gigabytes of memory. Hugepage support is required for the large memory pool allocation in DPDK [DPDK Project, 2010].

In summary, DPDK's memory management is very important to achieve high performance for network applications due to the following three main reasons: (1) NUMA-aware memory allocation ensures that CPU cores only access local memory, so that the memory access latency can be minimized; (2) hugepages can significantly reduce TLB misses as it allows to use less number of TLB entries for the same amount of memory; (3) as modern memory controllers have several memory channels that can load or store data in parallel, DPDK minimizes the risk of one memory channel being the bottleneck by spreading the addresses of objects among memory channels.

2.3.3 Burst Packet I/O

DPDK allows network applications to receive and send packets in batches, thus it reduces the per-packet cost of accessing and updating the queue. Generally speaking, there is a trade-off between throughput and latency: when improving the throughput, one also expects to have a larger average end-to-end latency. However, the burst packet I/O only adds trivial delay because the burst size is small compared to the packet processing rate.

2.4 Acceleration with Modern CPUs

This section focuses on software packet processing acceleration techniques, which are motivated by the fact that the performance gap between CPU and memory is increasing. To address this performance issue, we have to hide memory access latency. We take a close look at the CPU cache hierarchy as well as several hardware features available in modern CPUs, including memory prefetching, out-of-order execution, etc.

Figure 2·4 shows the single processor performance projections against the historical performance improvement in time to access main memory [Hennessy and Patterson, 2017]. Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the DRAM access latency, is plotted over time. As can be seen, the performance gap between CPU and memory remains very large. Although the single-core bandwidth has grown more slowly in recent years, the gap between CPU memory demand and DRAM bandwidth continues to grow as the numbers of cores grow in modern CPUs. The key problem is how to reduce *memory-access stalls* in programming. To accelerate network applications, we heavily rely on the advancement of modern CPUs, so let us first take a look at some features available in modern CPUs.

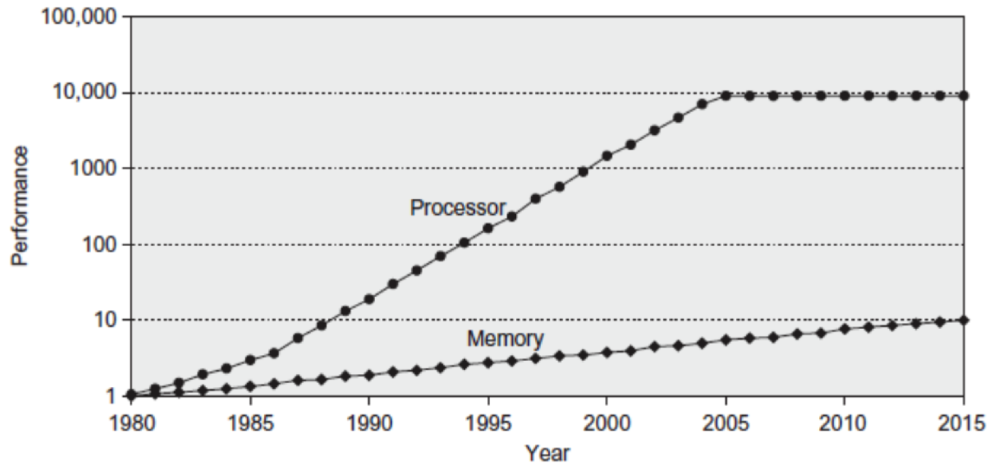


Figure 2·4: Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. [Hennessy and Patterson, 2017]

2.4.1 CPU Cache Hierarchy

CPUs have a hierarchical memory system, and faster memory has smaller size as they are more expensive. Typically, CPUs have three levels of cache: (1) a first-level instruction cache, and a first-level data cache (L1D); (2) a second level (L2) cache, in each CPU core; (3) a shared last level cache (LLC) among all cores in a physical processor package. Note that, the L1D cache may be shared by two logical processors if the processor support Intel HyperThreading Technology [Intel, 2019a], which allows a single CPU core to run a small number of hardware threads (typically two). This enables fast context switches to another hardware thread when one thread is stalled on memory [Kalia et al., 2015]. The L2 cache is shared by instructions and data.

Whenever the CPU issues a memory load, it first checks whether the data is in the L1D cache or not. If the data is in the L1D cache, then it accesses the data directly. Otherwise, a memory-access stall occurs. In this case, the Line Fill Buffers (LFBs) are checked to see if there is a memory request for the same cache line. If not, a new memory request is created, an empty LFB is allocated to track the status of the request, and the request itself gets forwarded to the L2 cache. The LFB can manage up to 10 requests of missing cache lines simultaneously, so the L1D can handle up to 10 outstanding cache misses and continue to service incoming stores and loads [Intel, 2019a]. If the requested address is not in the L2 cache, the request is then forwarded to the LLC. Finally, if the address is not in the LLC, the request goes to the memory controller and subsequently to the main memory.

Table 2.1: Intel CPU cache access latency (cycles) in different microarchitectures.

	Broadwell	Skylake Server	Skylake	Haswell	Sandy Bridge
L1 Cache	4 – 6	4 – 6	4	4	4
L2 Cache	12	14	12	11	12
LLC	50 – 60	50 – 70	44	34	26 – 31

After knowing how the CPU cache hierarchy works, let us take a look at the access latency as well as the size of the memory components. We take Intel CPUs for example. Table 2.1 shows the cache access latency of the Intel CPUs with different microarchitectures [Intel, 2019a]. The latency of the main memory access is 182 cycles [Psaropoulos et al., 2019]. As we can see, the processors' L1 access time is around 40$\times$ faster than the main memory, while L2 and L3 access latency is around 15$\times$ and 5$\times$ faster than the main memory access time, respectively. Typically, the access time of L1, L2, L3, and main memory has similar trends across the latest CPU architectures [Alipourfard et al., 2018]. For cache sizes, the L1 cache has 32KB (up to 64KB) capacity, L2 cache has 256KB (up to 1MB) capacity, and L3 cache has up to 2.5MB/core capacity [Intel, 2019a].

In summary, CPUs have much higher performance than memory accesses, and CPU may get stalled while waiting for loading data structures related to the current processing packet in network applications. To address this performance bottleneck, we can take advantage of three important features in CPUs as discussed in the following sections: hardware context switching, memory prefetching and out-of-order execution.

2.4.2 Hardware context switching

Historically, several hardware-based solutions have been proposed to hide latency. Two representative solutions are the Tera MTA-2 system [Alverson et al., 1990] and simultaneous multi-threading (SMT) [Tullsen et al., 1995, Tullsen et al., 1996].

The custom Tera MTA-2 system uses interleaved multi-threading to hide latencies without caches. The key insight is that it switches between threads to hide latency, and uses a deeply pipelined memory system to handle many simultaneous requests [Snavely et al., 1998]. However, it has relatively low single-threaded performance, which was not cost-effective. Later, a new model, named Cray XMT [Feo et al., 2005], was designed to reduce its cost, however, it still only has narrow range of applicability.

SMT uses thread-level parallelism to hide long-latency events in a processor, which has broad applications. The key insight behind SMT is that register renaming and dynamic scheduling allow multiple instructions from independent threads to be executed without regard to the dependences among them. Many processors support SMT. However, in practice, the existing implementations of SMT offer only two to four contexts with fetching and issue from only one, and up to four issues per clock. For instance, Intel i7 920, Intel Xeon E7 and Fujitsu SPARC64 X+ support SMT with two threads per core. As a matter of fact, the low limit on number of hardware threads per core is too small for many applications. In [Jonathan et al., 2018], we see that some data structures need around 200 tasks per thread to achieve the best performance. Moreover, as stated in [Hennessy and Patterson, 2017]: "the dominant portion of the increase in transistor count goes to increasing the caches and the number of cores per die." The number of hardware threads per core will remain small. Therefore, the gain from SMT is likely to remain modest for now and in the near future. Another challenge is performance degradation. The characterization of a key-value store (KVS) system in [Li et al., 2015] shows that Intel Hyper-threading technology with 2-way SMT causes a 24% throughput degradation with the full system setup. Additionally, we see performance degradation in the SPECFP benchmarks when they run in SMT mode [Hennessy and Patterson, 2017, Section 5.7].

Some other hardware-based multi-threading solutions, such as Cyclops [Almási et al., 2003] and GPUs [Fatahalian and Houston, 2008], have similar issues in practice. Therefore, CPUs with only available hardware context switching cannot hide all the memory access latency to achieve the highest possible throughput. In the following sections, we will discuss other important CPU features for further throughput improvement.

2.4.3 Memory Prefetching

When applications need to access large data structures kept in DRAM, modern CPUs have special hardware to prefetch memory locations, which allows the applications to fetch data ahead of demand to the L1D cache and hide memory latency by removing loads from the critical path. Developers can use hardware prefetching, software prefetching, or combination of the two. Software-managed prefetching allows the developers to explicitly tell the CPU to prefetch a memory location into a given level of cache. Table 2.2 shows the implementation details of prefetch hint instructions on popular Intel processors. “N” represents the prefetched data will not be inserted into that level of cache, while “Y” means the prefetched data will be inserted into that level of cache.

Memory prefetching can be enhanced by processing the packets in batch. The general idea is to process packets in a burst, thus reducing the per-packet packet processing cost. As modern CPUs can have multiple memory loads in flight at the same time, batched packet processing allows one to issue multiple memory accesses over the batch of the packets at the same time instead of only issuing memory accesses for a single packet. Thus, batching

Table 2.2: Implementation details of prefetch hint instructions.

Instruction	L1	L2	L3	Remarks
PrefetchT0	Y	Y	Y	Temporal data with respect to all level caches
PrefetchT1	N	Y	Y	Temporal data with respect to the L2 and L3 caches
PrefetchT2	N	Y	Y	Temporal data with respect to the L2 and L3 caches
PrefetchNTA	Y	N	Y	Non-temporal data with respect to the L1 and L3 caches

further reduces the per-packet memory access latency.

2.4.4 Other Features in CPUs

Besides the hugepages support in modern CPUs (see discussion in §2.3.2), two advanced modern CPU features discussed below are very important for applications to achieve high performance.

Out-of-order Execution: the out-of-order (OOO) engine in modern CPUs can detect dependency chains of the micro-ops and send them to execution out-of-order, while maintaining the correct data flow [Intel, 2019a]. When a dependency chain is waiting for a resource, such as a second-level data cache line, it sends micro-ops from another chain to the execution core. After a CPU issues a memory access, out-of-order execution allows the CPU to continue executing independent instructions while awaiting the completion of a cache miss, thus hiding all or part of the cache miss penalty. However, as each core can only maintain up to ten outstanding cache misses [Kalia et al., 2015], the OOO execution cannot efficiently go much wider than several instructions. Nonetheless, OOO execution increases the overall rate of instructions executed per cycle (IPC).

Intel Data Direct I/O (DDIO) Technology: Intel DDIO [Intel, 2012] is a feature introduced with the Intel Xeon E5 and E7 processor family. With the improvement of the CPU cache hierarchy, the LLC is now more than 20 MB, which is no longer a scarce resource. DDIO makes the processor cache the primary destination and source of I/O data, and it helps the NICs to deliver increased bandwidth, lower latency, and reduced power consumption by injecting packets directly into the processor's LLC. Then, the CPU can directly access the packet data without going to main memory.

2.5 Related Work on Memory Prefetching

In this section, we introduce several memory prefetching techniques, which are key approaches to hide memory latency by overlapping processor computations with memory accesses. In high level, there are two types of memory prefetching techniques: hardware prefetching and software prefetching, and both techniques have been studied in detail in the past. We will briefly discuss the advantages and disadvantages of each technique and the related ongoing research in this area. Interested readers can read the recent survey paper with a comprehensive discussion [Mittal, 2016]. Then, we focus on two state-of-the-art software prefetching techniques: group prefetching and coroutines-based prefetching. Especially, in the last two years, coroutines were proposed to hide memory access latency for pointer-chasing data structures that are heavily used in database systems. We will bring this technique to optimize a complex networking system in Chapter 4, hoping that the networking community can become aware of the coroutines-based prefetching.

2.5.1 Hardware vs. Software Prefetching

Hardware Prefetching

The hardware-based solutions [Smith, 1982, Fu et al., 1992, Joseph and Grunwald, 1999, Nesbit et al., 2004, Nesbit and Smith, 2004, Somogyi et al., 2006] automatically bring cache lines into the caches based on prior data misses. Hardware prefetches are handled dynamically at run-time without compiler intervention. However, it requires some regularity in the data access patterns. For example, if the access stride is not constant, the hardware prefetcher can hide memory latency if the strides of two successive cache misses are less than the trigger threshold distance [Intel, 2019a]. Moreover, as the hardware prefetcher requires a couple of misses before it starts operating, it is not effective for short streams. The hardware prefetchers may also waste bus bandwidth (1) when the application's memory traffic has significant portions with strides of cache misses greater than the trigger distance threshold; (2) it generates requests for data beyond the end of an array.

In summary, the effectiveness of the hardware prefetching depends on the application's access patterns. Specifically, an application with a good temporal locality will benefit greatly from the automatic hardware prefetcher. Some recent research efforts focus on applying machine learning [Liao et al., 2009, Rahman et al., 2015, Braun and Litz, 2019] to improve the performance of data prefetchers. Nonetheless, the Intel CPUs still only support simple hardware prefetchers, such as streaming, stride, and spatial prefetchers [Intel, 2019a].

Software Prefetching

Software-based solutions for prefetching [Porterfield, 1989, Klaiber and Levy, 1991, Callahan et al., 1991, Mowry and Gupta, 1991, Mowry et al., 1994] require a programmer to perform static program analysis and to manually insert PREFETCH hint instructions to handle the cache misses. These instructions are hints to bring a cache line of data into various levels and modes in the cache hierarchy. Software prefetching can handle irregular access patterns which do not trigger the hardware prefetcher, and they can use less bus bandwidth than hardware prefetching. However, software prefetching can introduce some non-negligible execution overhead due to the extra prefetch instructions and that some useful prefetching cannot be uncovered at run-time. Moreover, it may require some code structure change to provide timely prefetching. Otherwise, the prefetching can be either too late or too early, making it less useful or even harmful. Some ongoing research efforts focus on the interactions between software and hardware prefetches [Wang et al., 2003, Guttman et al., 2015], and some focus on compiler support [Mehta et al., 2014].

Several papers [Chen and Baer, 1994, Zucker et al., 2000, Lee et al., 2012] evaluated the performance of various hardware-based and software-based prefetches under different environmental settings. In summary, even though hardware prefetching has been improved significantly, the software prefetches are still needed in the following situations: (1) a large number of streams, because hardware prefetches do not perform well due to its limited hardware resources (*e.g.*, Intel CPUs can detect and maintain up to 32 streams of data accesses [Intel, 2019a]); (2) short streams, as hardware prefetchers require training time to detect the direction and distance of a stream or stride; (3) irregular memory accesses and pointer chasing [Fan et al., 2013], as memory references for complex access patterns are difficult to predict using hardware prefetches.

Three state-of-the-art software prefetching techniques have been proposed recently to hide memory latency: group prefetching (GP) [Chen et al., 2007], asynchronous memory access chaining (AMAC) [Kocberber et al., 2015], and coroutines [Psaropoulos et al., 2017, Jonathan et al., 2018, Psaropoulos et al., 2019]. The key insight behind these techniques is to process packets in batch and try to hide memory latency by separating the *arithmetic* operations and *memory access* operations, so that they can use memory prefetching to issue a request to load a given memory location into cache ahead of demand when processing a packet. After issuing the request, thanks to the out-of-order execution, the CPU can still execute independent instructions (*e.g.*, hash computations) for another packet while not waiting for the request to complete. Thus, the applications can still do useful computation while loading from memory, which can cut most of the memory access latency (if not all). The main difference between these software prefetching techniques is the fast context switching scheme. As pointed out in [Jonathan et al., 2018], coroutines-based prefetching has most of the performance benefits of AMAC approach while maintaining very high developer productivity, not possible in AMAC. A coroutine is a control abstraction that generalizes subroutines by allowing suspension and resumption. Thus, a coroutine can be suspended, transfer control to other coroutines, and resume later. The details of coroutines will be further discussed in §2.5.3. In the rest of this section, we will present how GP and coroutine-based prefetching address the performance issue by using an example of hash table lookup.

2.5.2 Group Prefetching

Group prefetching was originally proposed to hide memory latency for hash joins [Chen et al., 2007]. After that, it has been applied to several networking systems. For example, the software-based layer-2 switch - CuckooSwitch [Zhou et al., 2013] uses group prefetching to accelerate the lookup procedure of the cuckoo hash table. MICA [Lim et al., 2014] is an in-memory key-value storage system that uses multi-stage prefetching to interleave computation and memory access.

Algorithm 1: Group prefetching-based hash table lookup

```
Input: hash table data structure: $h$
Input: batch size: $B$
Input: hash key array: $K[0 \cdots B - 1]$
Output: value array: $V[0 \cdots B - 1]$
init V[B]              /* output placeholder */
init entry_idx[B]      /* entry index placeholder */
init *v_ptr[B]         /* value pointer placeholder */
for $i \leftarrow 0$ to $B-1$ do
    entry_idx[i] = hash(K[i])
    prefetch (&h[entry_idx[i]])
end
for $i \leftarrow 0$ to $B-1$ do
    v_ptr[i] = h[entry_idx[i]].v_ptr
    prefetch (v_ptr[i])
end
for $i \leftarrow 0$ to $B-1$ do
    if v_ptr[i] != null then
        V[i] = *v_ptr[i]
    end
end
return V
```

Let us take a simple hash table lookup for example. For the sake of presentation, we assume that the hash table uses a single hash function and there are no hash collisions. Each entry in the hash table stores the pointer to the corresponding value instead of the value itself. Basically, there are three steps for each lookup: hash computation, accessing the hash table entry to get the value pointer, and finally accessing the value. Algorithm 1 shows the pseudocode of the group prefetching-based hash table lookup. Instead of accessing the hash table entry directly after computing the hash for the given key, it issues a memory prefetching for the hash table entry and proceeds to compute the hash for the remaining lookup keys (Lines $4 - 7$). Therefore, the CPU does not get stalled on memory accesses to the entry while continuing independent instructions (hash computation and prefetch instructions). Then, it issues memory prefetching for the value pointers of each hash table entry (Lines $8 - 11$). Finally, it accesses and stores the values (Lines $12 - 16$).

From the above example, we know that group prefetching executes each code stage for the whole group of packets before moving to the next code stage. Thus, it does not maintain any states for each packet, which minimizes the overhead added to the program. However, it requires heavy code changes: programmers have to conduct static code analysis, divide the code linearly into stages (difficult for programs with complex control flows), and then manually insert the prefetch instructions into the code. Another major disadvantage of group prefetching is that it cannot start to process new packets even when some packets are done in the group.

2.5.3 Coroutines

Coroutines were proposed to simplify the cooperation between the lexical and syntactical analyzers in a COBOL compiler [Conway, 1963] in 1963. After that, coroutines have been widely used in different contexts, such as simulation, text processing, concurrent programming [Moura and Ierusalimschy, 2009]. Specifically, they are used to implement many program components such as cooperative tasks, exceptions, generators, iterators, etc. Recently, several research efforts [Psaropoulos et al., 2017, Jonathan et al., 2018, Psaropoulos et al., 2019] pioneered the use of coroutines for hiding memory latency for pointer-chasing data structures that are heavily used in database systems.

A coroutine [Conway, 1963] is a control abstraction that generalizes subroutines by allowing suspension and resumption. When a subroutine is invoked, it starts the execution, and can only run to completion. Thus, a subroutine instance only returns once, and does not hold state between invocations. A coroutine can be suspended and transfer control to other coroutines. Later, the coroutine can resume its execution after returning to the point where it left off. Thus, a coroutine instance needs to hold a state between successive calls. Although coroutines have been around for more than 50 years, some programming languages did not support them until recently. For example, the technical specification for coroutines has been merged into C++ [ISO/IEC, 2017].

Algorithm 2: Coroutines-based hash table lookup

```
Input: hash table data structure: h
Input: hash key: key
Output: value
init value
init entry_idx
init v_ptr
entry_idx = hash(key)
prefetch (&h[entry_idx)
co_suspend()
v_ptr = h[entry_idx].v_ptr
prefetch (v_ptr)
co_suspend()
if v_ptr != null then
    value = *v_ptr[i]
end
return value
```

We use coroutines extensively to optimize our Gatekeeper system in chapter 4, which is written mainly in C programming language. However, the C programming language does not have direct support for coroutines. In fact, there are a lot of C libraries for coroutines with different APIs. For the sake of presentation, we use a custom function call **co_suspend()** to transfer the control to other coroutines instead of using APIs from a specific library. For now, let us assume there is a program that maintains all the coroutines and manages the transfer logic among them. We will discuss the implementation details in Chapter 4.

Algorithm 2 shows the pseudocode for a coroutine-based implementation of a hash table lookup. On initializing a new key lookup, we initialize a coroutine and store it in the buffer managed by a program. The coroutine has two suspension points (Lines 6 and 9), where the coroutine suspends itself and returns back to the caller. When a coroutine suspends itself, the caller will retrieve the next coroutine from the buffer and resume it. When a coroutine finishes its execution − *i.e.*, it has found the matching key, or there are no more entries to fetch − it returns back to the caller, allowing the caller to replace it with a new coroutine, for the next key lookup. As can be seen, the coroutines-based prefetching does not change the code structure, and the only thing needed is to add a **prefetch** instruction whenever there is an expensive memory access, and after that the coroutine suspends itself. One can easily extend this function with an additional parameter indicating whether the memory prefetching is enabled or not. If disabled, the function will skip the memory prefetching and coroutine suspension, then it does exactly the same thing with the original function. Later, we show that coroutines-based prefetching improves the performance of Gatekeeper system by up to $1.9\times$ in Chapter 4.

2.6 The Future of NICs

Currently, most network applications rely on host CPUs to deal with the high-speed network links, and they achieve high-speed packet processing by bypassing the OS networking stack and running cores in poll-mode. For example, in an Intel DPDK performance report, the achieved IO throughput on a single CPU core is over 40 million packets per second [DPDK Project, 2019]. With only a few CPU cores, the multi-threaded DPDK applications can handle 100 Gbps links (148.88 Mpps for 64-byte packets). As modern servers typically have a large number of CPU cores (*e.g.*, up to hundreds of cores), the IO overhead with optimizations is small compared to the available computing resources.

However, several factors may limit its applicability. First and foremost, with the slowdown of Moore's law, the increase of CPU performance cannot keep up with the ever-growing network link speed. Therefore, with faster network links (*e.g.*, 200 Gbps, 400 Gbps, 1.6 Tbps), the network applications will need more and more precious CPU cores to achieve line-speed packet processing, which will become quite challenging. Moreover, as network applications need to support more and more features, the packet processing overhead becomes larger. Thus, the maximum packet rate that a single CPU core can handle will decrease with more features, which also requires more CPU cores. Finally, cloud providers would like to save as many precious CPU cores as possible, so that they can provide more computing resources that can run general-purpose applications to their customers [Firestone et al., 2018]. With the end of Moore's law, the only path left to improve energy-performance-cost is specialization [Hennessy and Patterson, 2017]. In the future, we have to rely on domain-specific hardware (*e.g.*, SmartNICs) to perform only a narrow range of computations for network tasks, and they should do much better than CPUs.

In the following sections, we will introduce several hardware choices for packet processing, discuss the advantages and disadvantages of these hardware solutions. Moreover, we introduce SmartNICs, classify them into three types based on the hardware choice for packet processing, and then dive into the recent work on SmartNICs. Finally, we discuss the future of these NICs.

2.6.1 Overview of Hardware Choices

In this section, we give an overview of various hardware choices for packet processing: CPU, GPU, FPGA (Field-Programmable Gate Array), and ASIC (Application-Specific Integrated Circuit). Each one has its own advantages and disadvantages. Table 2.3 compares them in terms of programmability, latency, energy efficiency, and development cost. This table is summarized with our own experiential knowledge.

Table 2.3: A comparison of various types of hardware.

	Programmability	Latency	Energy efficiency	Development cost
CPU	Easy	Medium	Low	Low
GPU	Medium	High	Medium	Medium
FPGA	Hard	Low	High	High
ASIC	Limited	Low	Highest	Medium

CPUs are optimized for sequential code through instruction-level parallelism and many of the available transistors are assigned to non-computational tasks like caching and branch prediction [Owens et al., 2007]. Therefore, CPUs are very suitable for memory-intensive network applications, especially with good cache locality. For example, in [Yang et al., 2018b], our SAIL implemented on CPU outperforms its implementation on FPGAs, GPUs and many-core platforms. Moreover, CPUs support a large number of programming languages, such as C, C++, Java, etc. The development cost is low.

Compared to CPUs, the main strengths of GPUs are vectorization through massive concurrency and memory latency hiding through fast hardware context switching [Kalia et al., 2015]. Specifically, GPUs use additional transistors for computation and achieve higher arithmetic intensity with the same transistor count [Owens et al., 2007]. Therefore, GPUs are very suitable for compute-intensive algorithms, such as cryptographic algorithms, that are used in network applications. However, GPUs introduce higher latency than CPUs [Cerović et al., 2018]. Newer technologies, such as Accelerated Processing Unit

(APU) [Go et al., 2017], put both CPUs and GPUs on the same die, which reduce the communication overhead between CPU and GPU and make GPUs suitable for a broader set of applications. Also, the programs written for CPUs have better backward compatibility than GPUs because the CPUs have better API consistency than GPUs [Cerović et al., 2018].

FPGAs have millions of Logic Elements (LEs) and thousands of DSP blocks for massive amount of parallelism. FPGAs are more energy efficient than GPUs and CPUs, however, they are less dense and more power hungry than ASICs [Gai, 2020]. Additionally, FPGAs provide deterministic timing, so that their latencies are one order of magnitude lower than GPUs [Cerović et al., 2018]. Although FPGAs can be entirely reprogrammed by downloading a new configuration file, the programmers need to write programs in a hardware description language (HDL), which requires a deep understanding of how hardware circuits and logic work. Also, debugging in FPGAs is very hard and the productivity is low. Compared to CPUs, the degree of programmability as well as the hardware resources are still limited. Therefore, the development cost is high.

ASICs are custom design for specific tasks; they can provide the highest performance potential with highest energy efficiency. However, they have limited programmability, as they cannot be programmed to process general-purpose tasks. Typically, the ASIC vendors provide C-like abstractions that a typical programmer is familiar with, so the development cost is medium [Choi et al., 2019].

2.6.2 Network Acceleration using SmartNICs

SmartNICs were recently proposed to deal with the ever-growing link speed with lower cost. Gai in his latest book [Gai, 2020] pointed out one simple definition for SmartNIC: *a NIC that offloads processing tasks that the system CPU would normally handle*. There are three major types of SmartNICs to deal with high-speed networking: (1) System-on-Chip (SoC)-based NICs; (2) FPGAs; (3) ASIC-based NICs. In the following sections, we present recent research efforts on these NICs and briefly discuss the future of these NICs.

SoC-based NICs

For multicore SoC-based NICs, they use a sea of CPU cores with simple microarchitectures to process packets for cost-effectiveness. Common processor choices are ARM or MIPS cores with associated caches. Moreover, they have onboard DRAM, domain-specific accelerators, and programmable DMA engines. The easy programmability is the key advantage of this architecture, as the processors can run standard Linux distributions and can be programmed in any language, which is quite suitable for software programmers. However, the future of SoC-style network offload is still questionable as it needs a sea of cores to scale with the link speed [Firestone et al., 2018]. Even with the latest 7 nm ARM with 96 cores at 3 GHz can handle 100 Gbps link but not multiple 200/400 Gbps links [Gai, 2020]. Moreover, it can lead to poor latency and jitter with a standard OS, which aims to maximize throughput. The recent work on iPipe framework [Liu et al., 2019a] and E3 [Liu et al., 2019b] show that SoC-based SmartNICs can save the host CPU cores, lower the latency, and improve energy-efficiency, cost efficiency for distributed applications as well as microservices. Moreover, if applications can utilize the available network accelerators on the SoC-based SmartNICs, they can achieve performance comparable to FPGA-based ones [Liu et al., 2019a].

FPGAs

Recently, several research efforts try to offload network functions onto FPGA-based SmartNICs (*e.g.*, Floem [Phothilimthana et al., 2018], KV-Direct [Li et al., 2017], ClickNP [Li et al., 2016], AccelNet [Firestone et al., 2018]). Due to the limitations of FPGAs (see §2.6.1), they are applicable to a specific class of applications but not for all. For example, the distributed applications with complex data structures and algorithms cannot be realized efficiently on FPGA-based SmartNICs.

ASIC-based NICs

As the ASIC-based NICs are designed for specific purposes, they can provide the highest performance potential. However, the addition of new features or the modification of the existing ones will require the redesign of these NICs, and the turnaround time is a big concern [Firestone et al., 2018]. The advancement of several technologies are changing the landscape to make ASIC-based NICs a better solution: (1) embedded CPU cores are added to the ASIC-based NICs, so that they can provide programmability for new functionalities. Especially, they can be used to implement the management and control plane, which has less data to deal with; (2) the Programming Protocol-independent Packet Processors (P4) [Bosshart et al., 2014] architecture defines the data plane behavior of a network device, and it provides a programmable data plane that allows the programmers to introduce new features without having to redesign the switch ASICs. Although P4 targets at the network switches, there are some ongoing work to extend P4 to be better suited for the NIC architecture [Choi et al., 2019, Gai, 2020]; (3) Some recent research efforts [Dean, 2019] focus on the use of machine learning to learn to automatically generate high quality solutions for designing custom ASICs, which can potentially reduce the turnaround time of the ASCI design significantly.

2.6.3 Discussion

Recently, we are seeing the convergence of different types of SmartNICs driven by cloud computing. In my opinion, the ASIC-based NICs have the potential to provide the best energy-performance-cost solutions for many network applications. Basically, these SmartNICs will integrate different hardware resources on a board (*e.g.*, CPU cores, DRAM, DMA engines, network accelerators for fixed functions, ASIC-based data plane, FPGAs, etc.). With the advancement of the on-board hardware, SmartNICs will have more powerful CPUs, large caches (*e.g.*, around 30 MB LLC), large DRAM (more than 8 GB), etc.

With enough hardware resources, not only the whole network stack can be offloaded to the NICs, but also part or even all of the application logic can be offloaded, as some data structures and algorithms can be efficiently implemented on these NICs. Moreover, with the support for different types of hardware, SmartNICs could provide great flexibility, programmability as well as performance, and they will have broader applications. In terms of programmability, many solutions are proposed recently to extend P4 language for different platforms, such as P4FPGA [Wang et al., 2017] and P4GPU [Li and Luo, 2016]. P4 has the potential to become the programming language for various hardware solutions, lowering the development cost.

Some examples of SmartNICs are Mellanox BlueField [Mellanox, 2020] and Netronome Agilio LX [Netronome, 2018], in which a single port can deliver up to 100 Gbps throughput. SmartNICs avoid many overheads (*e.g.*, PCIe overhead), achieve lower latency and higher throughput, and potentially provide a better energy-performance-cost solution for both IP lookup and DDoS mitigation. Moreover, the high programmability of the P4 language allows us to utilize various hardware resources easily. Therefore, we would expect that these SmartNICs are good candidates to improve the performance of both IP lookup and DDoS mitigation. Some tasks in the two applications (*e.g.*, encapsulation/decapsulation, declining flows, limiting flows' bandwidth, etc.) can be offloaded to SmartNICs without involving the host CPUs. We will explore their design space using SmartNICs in §5.2.

Chapter 3

SAIL: A Splitting Approach to IP Lookup

This chapter drills into the well-studied IP lookup problem and follows the definition of IP addresses and their formats, including IPv4, IPv6 and CIDR, defined in §2.1. An ideal IP lookup algorithm should achieve small constant IP lookup time, and on-chip memory usage. However, no prior IP lookup algorithm achieves both requirements at the same time. In this chapter, we propose a new IP lookup algorithm called `SAIL` [Yang et al., 2014, Yang et al., 2018b], a splitting approach to IP lookup. One splitting is along the dimension of the lookup process, namely finding the longest matching prefix length and finding the next hop. The second splitting is along the dimension of prefix length, namely IP lookup on prefixes of length less than or equal to 24 and that longer than 24 for IPv4. Moreover, we propose a suite of algorithms for IP lookup based on our SAIL framework. We conducted extensive experiments to evaluate our algorithms using real FIBs and real traffic from a major ISP in China. Experimental results show that our SAIL algorithms are much faster than well-known IP lookup algorithms.

The success of our SAIL approach is derived from the hardware improvement of the modern CPU cache hierarchy. Basically, the amount of on-chip memory we have is already larger than the theoretical maximum value (4MB) that SAIL needs. So we can implement our lookup algorithm in a much more straightforward way than other methods could. In so doing, we avoid some of the complexities that the state-of-the-art algorithms had to put in to fit on-chip memory restrictions at the time they were proposed.

3.1 Introduction

With the development of the Internet, the size of Fowarding Information Base FIB (FIB) in backbone routers grows rapidly. According to the RIPE Network Coordination Centre, FIB sizes have become more than 700,000 entries as of 2017 [RIPE Network Coordination Centre, 2017]. At the same time, cloud computing and network applications have driven the expectation on router throughput to the scale of 200 Gbps. The fast growth of FIB sizes and throughput demands bring significant challenges to IP lookup. An ideal IP lookup algorithm should satisfy the following two challenging requirements. First, *IP lookup time should meet wire speed, yet remain constant, as FIB sizes grow.* IP lookup time is per packet cost and should be optimized to the extreme to meet wire speed. Second, *on-chip memory usage should meet capacity constraints, yet remain constant, as FIB sizes grow*. On-chip memory (such as CPU cache and FPGA block RAM) is about 10 times faster than off-chip DRAM [Wang and Hamdi, 2008], but is limited in size (in the scale of a few MB) and much more expensive than DRAM; furthermore, as on-chip memory technologies advance, its sizes do not grow much as compared to DRAM. With network virtualization, a virtual router could have hundreds of virtual FIBs, which makes fast FIB lookup with small on-chip memory even more critical. Without satisfying these requirements, router performance will degrade as the FIB size grows, and router hardware will have to be upgraded periodically.

3.1.1 Summary and Limitations of Prior Art

IP lookup has long been a core networking issue and various schemes have been proposed. However, none of them satisfies the two requirements of both constant lookup time and constant on-chip memory usage. Some algorithms can achieve constant IP lookup time, such as TCAM based schemes [Zane et al., 2003, Zheng et al., 2006] and FPGA based schemes [Fadishei et al., 2005, Le et al., 2008], but their on-chip memory usage will grow quickly as FIB sizes grow. Some algorithms, such as full-expansion [Crescenzi et al., 1999] and DIR-24-8 [Gupta et al., 1998], can achieve constant memory usage by simply pushing all prefixes to levels 24 and 32, but even the lookup table for level 24 alone is too large to be stored in on-chip memory.

3.1.2 Proposed SAIL Approach

In this chapter, we present SAIL, a Splitting Approach to IP Lookup. We split the IP lookup problem along two dimensions as illustrated in Figure 3·1. First, we split the IP lookup problem into two sub-problems along the dimension of the lookup process: finding the prefix length (*i.e.*, finding the length of the longest prefix that matches the given IP address) and finding the next hop (*i.e.*, finding the next hop of this longest matched prefix). This splitting gives us the opportunity of solving the prefix length problem in on-chip memory and the next hop problem in off-chip memory. Furthermore, since on-chip and off-chip memory are two entities, this splitting allows us to potentially pipeline the processes of finding the prefix length and the next hop.

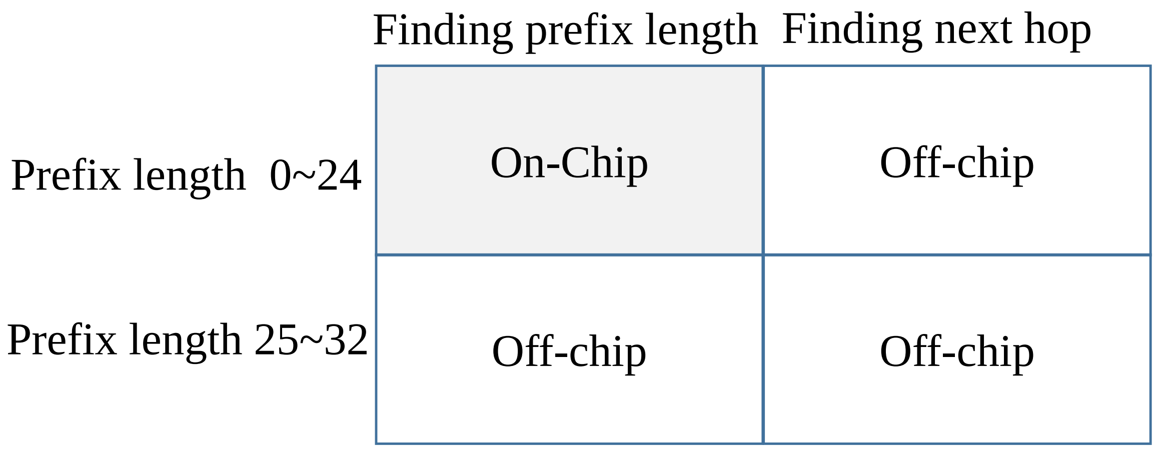


Figure 3·1: Two-dimensional splitting of IP lookup for IPv4

Second, we split the IP lookup problem into two sub-problems along the dimension of prefix length: length $\leqslant 24$ and length $\geqslant 25$. This splitting is based on our key observation that on backbone routers, for almost all traffic, the longest matching prefix has a length $\leqslant$ 24. This intuitively makes sense because typically backbone routers do not directly connect to small subnets whose prefixes are longer than 24. Our observation may not hold for edge routers, and the edge routers could have more long prefixes. However, even if all prefixes are longer prefixes, the worst case performance of our algorithm is still two off-chip memory accesses, and the on-chip memory usage is still bounded $\leqslant$ 4MB (See Table 3.2). The key benefit of this splitting is that we can focus on optimizing the IP lookup performance for traffic whose longest matching prefix length is $\leqslant 24$.

Some prior work performed splitting along the dimension of the lookup process or the dimension of prefix length; however, no existing work performed splitting along both dimensions. Dharmapurikar *et al.* proposed to split the IP lookup process into two sub-problems: finding the prefix length using Bloom filters and finding the next hop using hash tables [Dharmapurikar et al., 2003]. Pierluigi *et al.* and Gupta *et al.* proposed to split IP prefixes into 24 and 32 based on the observation that 99.93% of the prefixes in a backbone router FIB has a length of less than or equal to 24 [Crescenzi et al., 1999, Gupta et al., 1998]. Note that our IP prefix splitting criteria is different because our splitting is based on traffic distribution and their splitting is based on prefix distribution.

3.1.3 Technical Challenges and Solutions

The first technical challenge is to *achieve small constant on-chip memory usage for any FIB size*. To address this challenge, we propose to find the longest matching prefix length for a given IP address using bitmaps. Given a set of prefixes in a FIB, for each prefix length i $(0 \leqslant i \leqslant 32)$, we first build a bitmap $B_i[0..2^i-1]$ whose initial values are all 0s. Then, for each prefix p, we let $B_i[|p|] = 1$ where $|p|$ denotes the binary value of the first i bits of p. Thus, for all prefixes of lengths 0$\sim$24 in any FIB, the total memory size for all bitmaps is $\sum_{i=0}^{24} 2^i = 4\text{MB}$, which is small enough to be stored in on-chip memory.

The second technical challenge is to *achieve small constant IP lookup time for any*

FIB size. To address this challenge, we classify the prefixes in the given FIB into two categories: those with length $\leqslant 24$ and those with length $\geqslant 25$. (1) For prefixes of length $\leqslant 24$, for each prefix length i ($0 \leqslant i \leqslant 24$), we build a next hop array $N_i[0..2^i - 1]$ in off-chip memory so that for each prefix p whose next hop is $n(p)$, we let $N_i[|p|] = n(p)$. Thus, given an IP address $\boldsymbol{a}$, we first find its prefix length using bitmaps in on-chip memory and find the next hop using one array lookup in off-chip memory. To find the prefix length using bitmps, for i from 24 to 0, we test whether $B_i[\boldsymbol{a} \gg (32 - i)] = 1$; once we find the first i so that $B_i[\boldsymbol{a} \gg (32 - i)] = 1$ holds, we know that the longest matching prefix length is i. Here $\boldsymbol{a} \gg (32 - i)$ means right shifting $\boldsymbol{a}$ by $32 - i$ bits. In this step, the maximum number of on-chip memory accesses is 25. To find the next hop, suppose the longest matching prefix length for $\boldsymbol{a}$ is i, we can find its next hop $N_i[\boldsymbol{a} \gg (32 - i)]$ by one off-chip memory access. (2) For prefixes of length $\geqslant 25$, many IP lookup schemes can be plugged into our SAIL framework. Possible schemes include TCAM (Ternary Content Addressable Memory), hash tables, and next-hop arrays. Choosing which scheme to deal with prefixes of length $\geqslant 25$ depends on design priorities, but have little impact on the overall IP lookup performance because most traffic hits prefixes of length $\leqslant 24$. For example, to bound the worst case lookup speed, we can use TCAM or next hop arrays. For next hop arrays, we can expand all prefixes of length between 25 and 31 to be 32, and then build a chunk ID (*i.e.*, offsets) array and a next hop array. Thus, the worst case lookup speed is two off-chip memory accesses.

The third technical challenge is to *handle multiple FIBs for virtual routers* with two even more challenging requirements: (1) Multi-FIB lookup time should meet wire speed yet remain constant as FIB sizes and FIB numbers grow. (2) On-chip memory usage should meet capacity constraints yet remain constant as FIB sizes and FIB numbers grow. To address this challenge, we overlay all FIB tries together so that all FIBs have the same bitmaps; furthermore, we overlay all next hop arrays together so that by the next hop index and the FIB index, we can uniquely identify the final next hop.

3.2 Related Work

As IP lookup is a core networking issue, much work has been done to improve its performance. We can categorize prior work into trie-based algorithms, Bloom filter based algorithms, range-based algorithms, TCAM-based algorithms, FPGA-based algorithms, GPU-based algorithms, and multi-FIB lookup algorithms.

Trie-based Algorithms: Trie-based algorithms use the trie structure directly as the lookup data structure or indirectly as an auxiliary data structure to facilitate IP lookup or FIB update. Example algorithms include binary trie [Ruiz-Sánchez et al., 2001], path-compressed trie [Sklower, 1991], k-stride multibit trie [Srinivasan and Varghese, 1999], full expansion/compression [Crescenzi et al., 1999], LC-trie [Nilsson and Karlsson, 1999], Tree Bitmap [Eatherton et al., 2004], priority trie [Lim et al., 2010], Lulea [Degermark et al., 1997], DIR-24-8 [Gupta et al., 1998], flashtrie [Bando and Chao, 2010], shapeGraph [Song et al., 2009], trie-folding [Rétvári et al., 2013], DPP [Le and Prasanna, 2011], OET [Huang et al., 2011], DTBM [Sahni and Lu, 2007], multi-stride compressed trie [Mittal, 2010], and poptrie [Asai and Ohara, 2015]. More algorithms can be found in the literature [Lim and Lee, 2011, Ruiz-Sánchez et al., 2001, Yang et al., 2012a, Yang et al., 2012b, Zhao et al., 2010]

Bloom Filter based Algorithms: Dharmapurikar *et al.* proposed the PBF algorithm where they use Bloom filters to first find the longest matching prefix length in on-chip memory and then use hash tables in off-chip memory to find the next hop [Dharmapurikar et al., 2003]. Lim *et al.* proposed to use one Bloom filter to find the longest matching prefix length [Lim et al., 2012]. These Bloom filter-based IP lookup algorithms cannot achieve constant lookup time because of false positives and hash collisions. Furthermore, to keep the same false positive rate, their on-chip memory sizes grow linearly with the increase of

FIB size. Several Bloom filter variants can be found in the literature [Dai et al., 2016b, Yang et al., 2017, Dai et al., 2018, Yang et al., 2016, Dai et al., 2016a].

Range-based Algorithms: Range-based algorithms are based on the observation that each prefix can be mapped into a range in level 32 of the trie. Example algorithms are binary search on prefix lengths [Waldvogel et al., 1997], binary range search [Ruiz-Sánchez et al., 2001], multiway range trees [Warkhede et al., 2004], and DXR [Zec et al., 2012].

TCAM-based Algorithms: TCAMs can compare an incoming IP address with all stored prefixes in parallel in hardware using one cycle, and thus can achieve constant lookup time. However, TCAM has very limited size (typically a few Mbs like on-chip memory sizes), consumes a huge amount of power due to the parallel search capability, generates a lot of heat, is very expensive, and difficult to update. Some schemes have been proposed to reduce power consumption by enabling only a few TCAM banks to participate in each lookup [Zane et al., 2003]. Some schemes use multiple TCAM chips to improve lookup speed [Panigrahy and Sharma, 2002, Zheng et al., 2006, Akhbarizadeh et al., 2006]. Shah *et al.* proposed to reduce the movement of prefixes for fast updating [Shah and Gupta, 2001].

FPGA-based Algorithms: There are two main issues to address for FPGA-based IP lookup algorithms: (1) how to store the whole FIB in on-chip memory, and (2) how to construct pipeline stages. Some early FPGA-based algorithms proposed to construct compact data structures in on-chip memory [Sangireddy et al., 2005, Meribout and Motomura, 2003]; however, these compact data structures make the lookup process complex and therefore increase the complexity of FPGA logics. For FPGA in general, the more complex the logics are, the lower the clock frequency will be. To improve lookup speed, Fadishei *et al.* proposed to only store a part of data structure in on-chip memory using hashing [Fadishei et al., 2005]. To balance stage sizes, some schemes have been proposed to adjust the trie structure by rotating some branches or exchanging some bits of the prefixes [Baboescu et al., 2005, Le et al., 2008, Pao et al., 2014].

GPU-based Algorithms: Leveraging the massive parallel processing capability of GPU, some schemes have been proposed to use GPU to accelerate IP lookup [Han et al., 2010, Zhao et al., 2010].

Multi-FIB Lookup Algorithms: The virtual router capability has been widely supported by commercial routers. A key issue in virtual routers is to perform IP lookup with multiple FIBs using limited on-chip memory. Several schemes have been proposed to compress multiple FIBs [Fu and Rexford, 2008, Luo et al., 2013, Song et al., 2010].

3.3 SAIL Basics

In this section, we present the basic version of our SAIL algorithms. Table 3.1 lists the symbols used in this thesis.

Table 3.1: Symbols used in the thesis

Symbol	Description
B_i	array for level i
N_i	next hop array for level i
C_i	chunk ID array for level i
BN_i	combined array of B_i and N_i
BCN_i	combined array of B_i, C_i and N_i
$\boldsymbol{a}$	IP address
v	trie node
p	prefix
$\|p\|$	value of the binary string in prefix p
$p(v)$	prefix represented by node v
$n(v)$	next hop of solid node v
l	trie level
$\boldsymbol{a}_{(i,j)}$	integer value of bit string of $\boldsymbol{a}$ from i-th bit to j-th bit

3.3.1 Splitting Lookup Process

We now describe how we split the lookup process for a given IP address into the two steps of finding its longest matching prefix length and finding the next hop. Given a FIB table, we first construct a trie. An example trie is in Figure 3·3(b). Based on whether a node represents a prefix in the FIB, there are two types of nodes: *solid nodes* and *empty nodes*.

A node is solid if and only if the prefix represented by the node is in the FIB. That is, a node is solid if and only if it has a next hop. A node is an empty node if and only if it has no next hop. Each solid node has a label denoting the next hop of the prefix represented by the node. For any node, its distance to the root is called its *level*. The level of a node locates a node vertically. Any trie constructed from a FIB has 33 levels. For each level i, we construct a array $B_i[0..2^i-1]$ of length 2^i, and the initial values are all 0s. At each level i $(0 \leqslant i \leqslant 32)$, for each node v, let $p(v)$ denote its corresponding prefix and $|p(v)|$ denote the value of the binary string part of the prefix (*e.g.*, $|11*| = 3$). If v is solid, then we assign $B_i[|p(v)|] = 1$; otherwise, we assign $B_i[|p(v)|] = 0$. Here $|p(v)|$ indicates the horizontal position of node v because if the trie is a complete binary tree then v is the $|p(v)|$-th node at level i. Figure 3·3(c) shows the s for levels 0 to 4 of the trie in 3·3(b). Taking B_3 for level 3 as an example, for the two solid nodes corresponding to prefixes `001*/3` and `111*/3`, we have $B_3[1] = 1$ and $B_3[7] = 1$. Given the 33 bitmaps $B_0, B_1, \cdots, B_{32}$ that we constructed from the trie for a FIB, for any given IP address $\boldsymbol{a}$, for i from 32 to 0, we test whether $B_i[\boldsymbol{a} \gg (32-i)] = 1$; once we find the first i that $B_i[\boldsymbol{a} \gg (32-i)] = 1$ holds, we know that the longest matching prefix length is i. Here $\boldsymbol{a} \gg (32-i)$ means right shifting $\boldsymbol{a}$ by $32-i$ bits. For each B_i, we construct a next hop array $N_i[0..2^i-1]$, whose initial values are all 0s. At each level i, for each prefix p of length i in the FIB, denoting the next hop of prefix p by $n(p)$, we assign $N_i[|p|] = n(p)$. Thus, for any given IP address $\boldsymbol{a}$, once we find its longest matching prefix length i, then we know its next hop is $N_i[\boldsymbol{a} \gg (32-i)]$.

3.3.2 Splitting Prefix Length

Based on our observation that almost all the traffic of backbone routers has the longest matching prefix length $\leqslant 24$ as illustrated in Figure 3·2, we split all prefixes in the given FIB into prefixes of length $\leqslant 24$, which we call *short prefixes* and prefixes of length $\geqslant 25$, which we call *long prefixes*. By this splitting, we want to store the bitmaps of prefixes of length $\leqslant 24$ in on-chip memory. However, given an IP address, because it may match a long

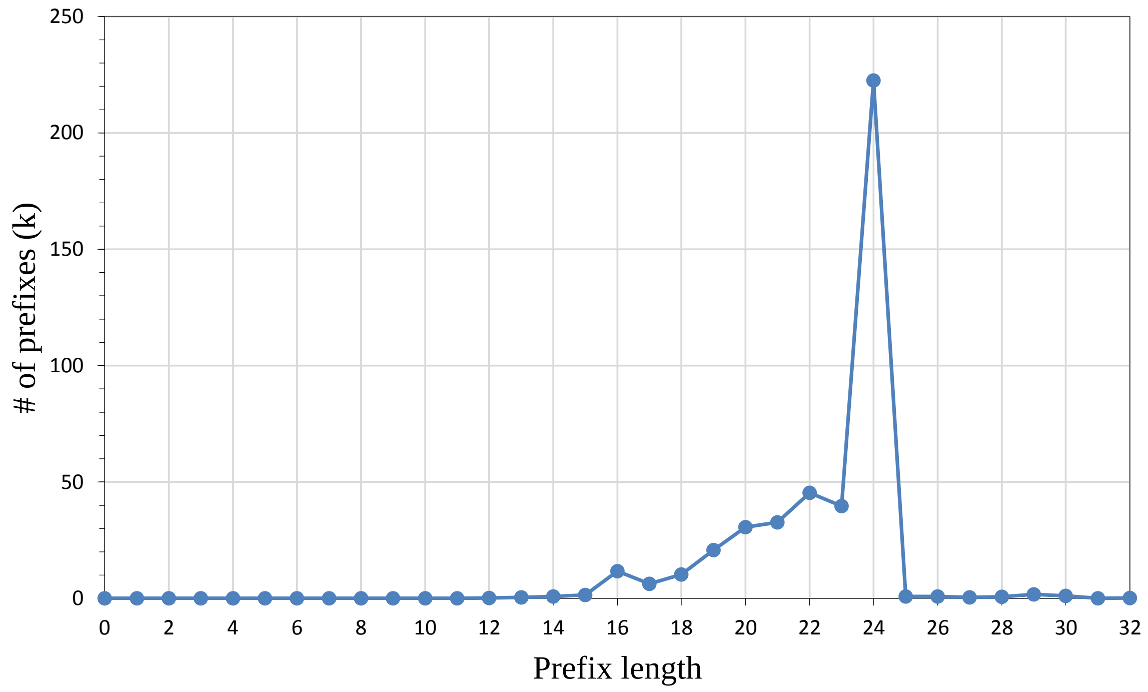
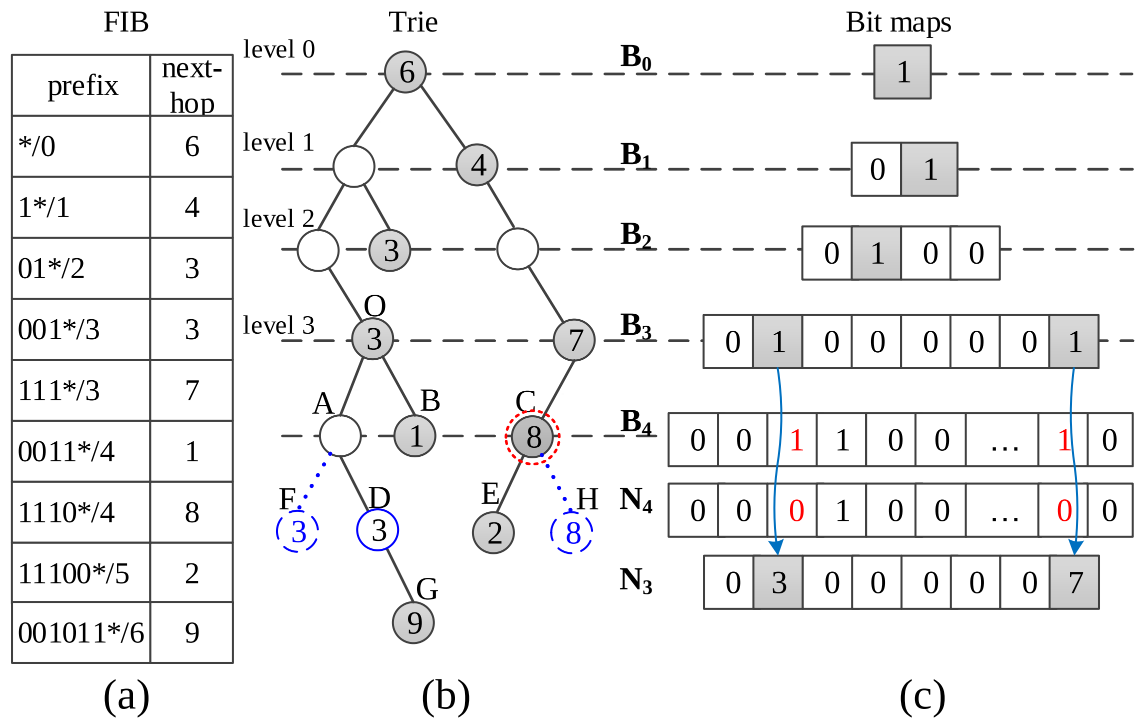


Figure 3·2: Prefix length distribution.



prefix	next-hop
*/0	6
1*/1	4
01*/2	3
001*/3	3
111*/3	7
0011*/4	1
1110*/4	8
11100*/5	2
001011*/6	9

Figure 3·3: Basic SAIL Algorithm.

prefix, it seems that we need to search among both short and long prefixes, which makes this splitting not too useful. We propose a technique called *pivot pushing* to address this issue. Our basic strategy is that for a given IP address, we first test its longest matching prefix length is within $[0,24]$ or $[25,32]$; thus, after this testing, we continue to search among either short prefixes or long prefixes, but not both. We call level 24 the *pivot level*.

Given a trie and a pivot level, the basic idea of pivot pushing is two-fold. First, for each internal solid node on the pivot level, we push its label (*i.e.*, the next hop) to a level below the pivot level. Second, for each internal empty nodes on the pivot level, we let it inherit the label of its nearest solid ancestor node, *i.e.*, the next hop of the first solid node along the path from this empty node to the root, and then push this inherited label to a level below the pivot level. In this thesis, we assume that the root always has a label, which is the default next hop. Thus, for any internal empty nodes on the pivot level, it always can inherit a label.

Given a trie and an IP address $\boldsymbol{a}$, traversing $\boldsymbol{a}$ from the root of the trie downward, for any internal or leaf node v that the traversal passes, we say $\boldsymbol{a}$ *passes* v. Based on the above concepts, we introduce Theorem 3.3.1.

Theorem 3.3.1 *Given a trie constructed from a FIB, after pivot pushing, for any IP address* $\boldsymbol{a}$*,* $\boldsymbol{a}$ *passes a node on the pivot level if and only if its longest matching prefix is on the pivot level or below.*

Proof: Given a trie and an IP address $\boldsymbol{a}$ that passes a node v on the pivot level, there are two cases: (1) v is a leaf node, and (2) v is an internal node. For the first case where v is a leaf node, then $\boldsymbol{a}$'s longest matching prefix is $p(v)$ (*i.e.*, the prefix represented by node v) and thus $\boldsymbol{a}$'s longest matching prefix is on the pivot level. For the second case where v is an internal node, because of pivot pushing, $\boldsymbol{a}$ must pass a solid node on a level below the pivot level, which means that $\boldsymbol{a}$'s longest matching prefix is below the pivot level. □

Based on Theorem 3.3.1, we construct the bitmap array for the pivot level l as follows: for any node v at level l, we assign $B_l[|p(v)|] = 1$; in other words, $B_l[i] = 0$ if and only if there is no node at level l that corresponds to the prefix denoted by i. Thus, given an IP address $\boldsymbol{a}$, $B_l[\boldsymbol{a} \gg (32-l)] = 1$ if and only if its longest matching prefix is on the pivot level or below. In SAIL, we choose level 24 to be the pivot level. By checking whether $B_{24}[\boldsymbol{a} \gg 8] = 1$, we know whether the longest matching prefix length is $\leqslant 23$ or $\geqslant 24$, which will guide us to search either up or down. Consider the example in Figure 3·3(b), taking level 4 as the pivot level, node C at level 4 is an internal solid node, pushing C to level 5 results in a new leaf solid node H with the same next hop as C. Note that after pushing node C down, node C becomes empty.

Given a pivot-pushed trie, we build a bitmap array and a next hop array for each level of 0 to 24 as above. Note that for any i $(0 \leqslant i \leqslant 23)$ and any j $(0 \leqslant j \leqslant 2^i - 1)$, $B_i[j] = 1$ if and only if there is a solid node at level i that corresponds to the prefix denoted by j; for level 24 and any j $(0 \leqslant j \leqslant 2^{24} - 1)$, $B_{24}[j] = 1$ if and only if there is a node, no matter solid or empty, at level 24 that corresponds to the prefix denoted by j. Note that $B_{24}[j] = 1$ and $N_{24}[j] > 0$ if and only if there is a leaf node at level 24 that corresponds to the prefix denoted by j, which means that the longest matching prefix length is 24. Note that $B_{24}[j] = 1$ and $N_{24}[j] = 0$ if and only if there is an empty node at level that corresponds to the prefix denoted by j, which means that the longest matching prefix length is $\geqslant 25$. Thus, given an IP address $\boldsymbol{a}$, if $B_{24}[\boldsymbol{a} \gg 8] = 0$, then we further check whether $B_{23}[\boldsymbol{a} \gg 9] = 1$, $B_{22}[\boldsymbol{a} \gg 10] = 1, \cdots, B_0[\boldsymbol{a} \gg 32] = 1$ until we find the first 1; if $B_{24}[\boldsymbol{a} \gg 8] = 1$, then we know $\boldsymbol{a}$'s longest matching prefix length is $\geqslant 24$ and further lookup its next hop in off-chip memory. It is easy to compute that the on-chip memory usage is fixed as $\sum_{i=0}^{24} 2^i = 4MB$. Consider the example in Figure 3·3. Given an address *001010*, as the pivot level is 4, since $B_4[001010 \gg 2] = B_4[0010] = B_4[2] = 1$ and $N_4[001010 \gg 2] = N_4[0010] = N_4[2] = 0$, then we know that the longest matching prefix length is longer than 4 and we will continue the search at levels below 4.

The pseudocode for the SAIL Basic, denoted by SAIL_B, is shown in Algorithm 3.

Algorithm 3: SAIL_B

```
Input: arrays: $B_0, B_1, \cdots, B_{24}$
Input: Next hop arrays: $N_0, N_1, \cdots, N_{24}$
Input: $a$: an IP address
Output: next hop of the longest matched prefix
if $B_{24}[a \gg 8] = 0$ then
    for $j = 23; j > 0; j--$ do
        if $B_j[a \gg (32 - j)] = 1$ then
            return $N_j[a \gg (32 - j)]$
        end
    end
end
else if $N_{24}[a \gg 8] > 0$ then
    return $N_{24}[a \gg 8]$
end
else
    lookup at levels $25 \sim 32$
end
```

There are multiple ways to handle prefixes of length $\geqslant 25$. Below we give one simple implementation using next hop arrays. Let the number of internal nodes at level 24 be n. We can push all solid nodes at levels 25$\sim$31 to level 32. Afterwards, the number of nodes at level 32 is $256 * n$ because each internal node at level 24 has a complete subtree with 256 leaf nodes, each of which is called a *chunk*. As typically $256 * n$ is much smaller than 2^{32} based on our experimental results on real FIBs, constructing a next hop array of size 2^{32} wastes too much memory; thus, we construct a next hop array N_{32} of size $256 * n$ for level 32. As we push all solid nodes at levels from 25 to 31 to level 32, we do not need bitmaps $B_{25}, B_{26}, \cdots, B_{32}$. Now consider the nodes at level 24. For each leaf node, its corresponding entry in B_{24} is 1 and its corresponding entry in next hop array N_{24} is the next hop of this node. For each internal node, its corresponding entry in B_{24} is 1 and its corresponding entry in next hop array N_{24} is the chunk ID in N_{32}, which multiplied by 256 plus the last 8 bits of the given IP address locates the next hop in N_{32}. To distinguish these two cases, we let the next hop be a positive number and the chunk ID to be a negative number whose absolute value is the real chunk ID value. To have negative values, chunk IDs are named starting from 1. With our pivot pushing technique, looking up an IP address $\boldsymbol{a}$ is simple: if $B_{24}[\boldsymbol{a} \gg 9] = 0$, then we know the longest matching prefix length is within $[0, 23]$ and further test whether $B_{23}[\boldsymbol{a} \gg 8] = 1$; if $B_{24}[\boldsymbol{a} \gg 8] = 1 \wedge N_{24}[\boldsymbol{a} \gg 8] > 0$, then we know that the longest matching prefix length is 24 and the next hop is $N_{24}[\boldsymbol{a} \gg 8]$; if $B_{24}[\boldsymbol{a} \gg 8] = 1 \wedge N_{24}[\boldsymbol{a} \gg 8] < 0$, then we know that the longest matching prefix length is longer than 24 and the next hop is $N_{32}[(|N_{24}[\boldsymbol{a} \gg 8]| - 1) * 256 + (\boldsymbol{a}\&255)]$. Notice that & is bitwise `AND` operator.

3.3.3 FIB Update for SAIL Basic

We now discuss how to adjust the lookup data structures when the given FIB changes. Note that the FIB update performance for levels 25$\sim$32 is less critical than that for levels 0$\sim$24. As most traffic hits levels 0$\sim$24, when the lookup data structures for levels 0$\sim$24 in on-chip memory change, no lookup can be performed before changes are finished and therefore may cause incorrect routing decisions. For the lookup data structures in off-chip memory, many IP lookup schemes that can handle prefixes of length $\geqslant 25$ can be plugged into SAIL_B. Different IP lookup schemes have different FIB update algorithms. Therefore, we focus on the update of data structures in on-chip memory.

For SAIL_B, updating the on-chip lookup data structures is simple: given an update prefix p with length of l, and its next hop is denoted by h. Note that $h = 0$ means to withdraw prefix p while $h > 0$ means to announce prefix p. If $l < 24$, we assign $B_l[|p|] = (h > 0)$ (*i.e.*, if $h > 0$, then we assign $B_l[|p|] = 1$; otherwise, we assign $B_l[|p|] = 0$). If $l = 24$, for the same update, we first locate the node in the trie, if it is an internal node, then B_{24} is kept unchanged; otherwise, we assign $B_{24}[|p|] = (h > 0)$. Note that for one FIB update, we may need to update both the on-chip and off-chip lookup data structures. A router typically maintains the trie data structure on the control plane and uses it to compute the changes that need to be made to off-chip lookup data structures. Because little traffic hits the off-chip lookup data structures for levels 25$\sim$32, updating the off-chip lookup data structures often can be performed in parallel with IP lookups on the on-chip data structures.

3.4 SAIL Optimization

In this section, we first present two optimization techniques of our SAIL algorithms, which favor the performance of FIB update and IP lookup, respectively. We use SAIL_U to denote SAIL with update oriented optimization, and SAIL_L to denote SAIL with lookup oriented optimization. Then, we extend SAIL_L to handle multiple FIBs.

3.4.1 Update Oriented Optimization

Data Structures & Lookup Process: In this optimization, by prefix expansion, we push all solid nodes at levels $0 \sim 5$ to level 6, all solid nodes at levels $7 \sim 11$ to level 12, all solid nodes at levels $13 \sim 17$ to level 18, and all solid nodes at levels $19 \sim 23$ to level 24. With this 4-level pushing, looking up an IP address $\boldsymbol{a}$ is the same as without this pushing, except that if $B_{24}[\boldsymbol{a} \gg 8] = 0$, then we further check whether $B_{18}[\boldsymbol{a} \gg 14] = 1$, $B_{12}[\boldsymbol{a} \gg 20] = 1$, and $B_6[\boldsymbol{a} \gg 26] = 1$ till we get the first 1. This 4-level pushing brings two benefits to IP lookup. First, it reduces the maximum number of array lookups in on-chip memory from 24 to 4. Second, it reduces the on-chip memory usage by 49.2% because we do not need to store $B_0 \sim B_5$, $B_7 \sim B_{11}$, $B_{13} \sim B_{17}$, and $B_{19} \sim B_{23}$.

FIB Update: While improving lookup speed and reducing on-chip memory usage, this pushing incurs no extra cost to the update of on-chip data structures. With this pushing, we still achieve one on-chip memory access per FIB update because of three reasons. First, for any FIB update, it at most affects $2^6 = 64$ bits due to the above pushing. Second, typically by each memory access we can read/write 64 bits using a 64-bit processor. Third, as the lengths of the four s B_6, B_{12}, B_{18}, and B_{24} are dividable by 64, the 64 bits that any FIB update needs to modify align well with word boundaries in on-chip memory. We implement each of these four s as an array of 64-bit unsigned integers; thus, for any FIB update, we only need to modify one such integer in one memory access.

3.4.2 Lookup Oriented Optimization

Data Structures: In SAIL_B and SAIL_U, the maximum numbers of on-chip memory accesses are 24 and 4, respectively. To further improve lookup speed, we need to push nodes to fewer number of levels. On one hand, the fewer number of levels means the fewer numbers of on-chip memory accesses, which means faster lookup. On the other hand, pushing levels $0 \sim 23$ to 24 incurs too large on-chip memory. Multi-levels of pivot pushing can be carried out independently. To trade-off between the number of on-chip memory accesses and the data structure size at each level, we choose two levels: one is between 0$\sim$23, and the other one is 24. In our experiments, the two levels are 16 and 24. In this optimization, by prefix expansion, we first push all solid nodes at levels $0 \sim 15$ to level 16; second, we push all internal nodes at level 16 and all solid nodes at levels 17$\sim$23 to level 24; third, we push all internal nodes at level 24 and all solid nodes at levels $25 \sim 31$ to level 32. We call this *3-level pushing*. For level 16, our data structure has three arrays: array $B_{16}[0..2^{16}-1]$, next hop array $N_{16}[0..2^{16}-1]$, and chunk ID array $C_{16}[0..2^{16}-1]$, where the chunk ID starts from 1. For level 24, our data structure has three arrays: array B_{24}, next hop array N_{24}, and chunk ID array C_{24}, where the size of each array is the number of internal nodes at level 16 times 2^8. For level 32, our data structure has one array: next hop array N_{32}, whose size is the number of internal nodes at level 24 times 2^8.

Lookup Process: Given an IP address $\boldsymbol{a}$, using $\boldsymbol{a}_{(i,j)}$ to denote the integer value of the bit string of $\boldsymbol{a}$ from the i-th bit to the j-th bit, we first check whether $B_{16}[\boldsymbol{a}_{(0,15)}] = 1$; if yes, then the $\boldsymbol{a}_{(0,15)}$-th node at level 16 is a solid node and thus the next hop for $\boldsymbol{a}$ is $N_{16}[\boldsymbol{a}_{(0,15)}]$; otherwise, then the $\boldsymbol{a}_{(0,15)}$-th node at level 16 is an empty node and thus we need to continue the search at level 24, where the index is computed as $(C_{16}[\boldsymbol{a}_{(0,15)}]-1)*2^8+\boldsymbol{a}_{(16,23)}$. At level 24, denoting $(C_{16}[\boldsymbol{a}_{(0,15)}]-1)*2^8+\boldsymbol{a}_{(16,23)}$ by i, the search process is similar

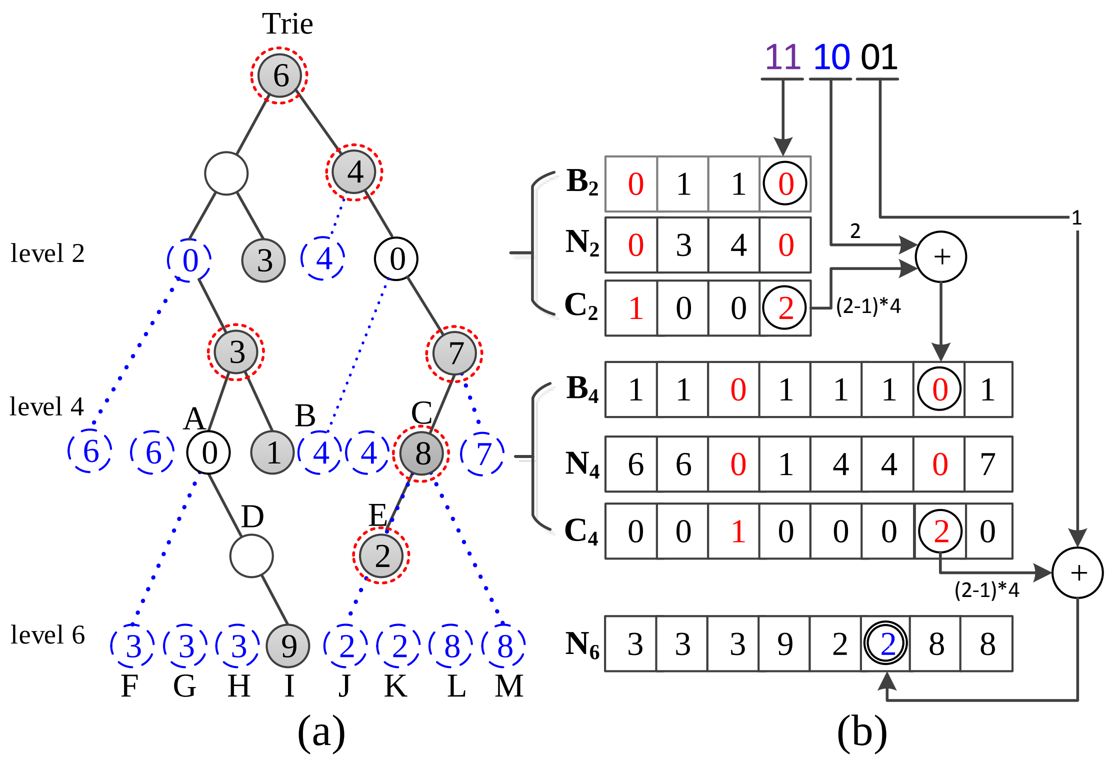


Figure 3·4: Example data structure of SAIL_L.

to level 16: we first check whether $B_{24}[i] = 1$, if yes, then the i-th node at level 24 is a solid node and thus the next hop for $\boldsymbol{a}$ is $N_{24}[i]$; otherwise, the i-th node at level 24 is an empty node and thus the next hop must be at level 32, to be more precise, the next hop is $(C_{24}[i]-1)*2^8+\boldsymbol{a}_{(24,31)}$. Figure 3·4 illustrates the above data structures and IP lookup process where the three pushed levels are 2, 4, and 6. The pseudocode for the lookup process of SAIL_L is in Algorithm 4, where we use the bitmaps, next hop arrays, and the chunk ID arrays as separate arrays for generality and simplicity.

Two-dimensional Splitting: The key difference between SAIL_L and prior IP lookup algorithms lies in its two-dimensional splitting. According to our two-dimensional splitting methodology, we should store the three arrays B_{16}, C_{16}, and B_{24} in on-chip memory and the other four arrays N_{16}, N_{24}, C_{24}, and N_{32} in off-chip memory as shown in Figure 3·5. We observe that for $0 \leqslant i \leqslant 2^{16}-1$, $B_{16}[i] = 0$ if and only if $N_{16}[i] = 0$, which holds if and only

Algorithm 4: SAIL_L

```
Input: arrays: $B_{16}, B_{24}$
Input: Next hop arrays: $N_{16}, N_{24}, N_{32}$
Input: Chunk ID arrays: $C_{16}, C_{24}$
Input: $\boldsymbol{a}$: an IP address
Output: the next hop of the longest matched prefix.
1  if $B_{16}[\boldsymbol{a} \gg 16] = 1$ then
2      return $N_{16}[\boldsymbol{a} \gg 16]$
3  end
4  $i \leftarrow ((C_{16}[\boldsymbol{a} \gg 16] - 1) \ll 8) + (\boldsymbol{a} \ll 16 \gg 24)$
5  else if $B_{24}[i] = 1$ then
6      return $N_{24}[i]$
7  end
8  else
9      return $N_{32}[((C_{24}[i] - 1) \ll 8) + (\boldsymbol{a}\&255)]$
10 end
```

if $C_{16}[i] \neq 0$. Thus, the three arrays of B_{16}, C_{16}, and N_{16} can be combined into one array denoted by BCN, where for $0 \leqslant i \leqslant 2^{16} - 1$, $BCN[i]_{(0,0)} = 1$ indicates that $BCN[i]_{(1,15)} = N_{16}[i]$ and $BCN[i]_{(0,0)} = 0$ indicates that $BCN[i]_{(1,15)} = C_{16}[i]$. Although in theory for $0 \leqslant i \leqslant 2^{16} - 1$, $C_{16}[i]$ needs 16 bits, practically, based on measurement from our real FIBs of backbone routers, 15 bits are enough for $C_{16}[i]$ and 8 bits for next hop; thus, $BCN[i]$ will be 16 bits exactly. For FPGA/ASIC platforms, we store BCN and B_{24} in on-chip memory and others in off-chip memory. For CPU/GPU/many-core platforms, because most lookups access both B_{24} and N_{24}, we do combine B_{24} and N_{24} to BN_{24} so as to improve caching behavior. $BN_{24}[i] = 0$ indicates that $B_{24}[i] = 0$, and we need to find the next hop in level 32; $BN_{24}[i] > 0$ indicates that the next hop is $BN_{24}[i] = N_{24}[i]$.

FIB Update: Given a FIB update of inserting/deleting/modifying a prefix, we first modify the trie that the router maintains on the control plane to make the trie equivalent to the updated FIB. Note that this trie is the one after the above 3-level pushing. Further note that FIB update messages are sent/received on the control plane where the pushed trie is maintained. Second, we perform the above 3-level pushing on the updated trie for the nodes

	Finding prefix length	Finding next hop
Prefix length 0~24	B_{16} , C_{16}, B_{24}	N_{16}, N_{24}
Prefix length 25~32	C_{24}	N_{32}

Figure 3·5: Memory management for SAIL_L.

affected by the update. Third, we modify the lookup data structures on the data plane to reflect the change of the pushed trie.

SAIL_L can perform FIB updates efficiently because of two reasons, although one FIB update may affect many trie nodes in the worst case. First, prior studies have shown that most FIB updates require only updates on a leaf node [Yang et al., 2012b]. Second, the modification on the lookup arrays (namely the arrays, next hop arrays, and the chunk ID arrays) is mostly continuous, *i.e.*, a block of a lookup array is modified. We can use the *memcpy* function in C to efficiently write 64 bits in one memory access on a 64-bit processor.

3.4.3 SAIL for Multiple FIBs

We now present our SAIL_M algorithm for handling multiple FIBs in virtual routers, which is an extension of SAIL_L. A router with virtual router capability (such as Cisco CRS-1/16) can be configured to run multiple routing instances where each instance has a FIB. If we build independent data structures for different FIBs, it will cost too much memory. One classic method is to merge all virtual FIBs into one, and then perform lookup on the merged one. Our design goal is to achieve constant on-chip memory and constant lookup time regardless of the number and the size of the virtual FIBs. No prior algorithm can achieve this goal. Next, we first show the classic merging method, and then show how to apply our SAIL framework to it.

Classic merging method of virtual FIBs: Given m FIBs $F_0, F_1, \cdots, F_{m-1}$, first, for each F_i $(0 \leqslant i \leqslant m-1)$, for each prefix p in F_i, we change its next hop $n_i(p)$ to a pair $(i, n_i(p))$. Let $F_0', F_1', \cdots$, and F_{m-1}' denote the resulting FIBs. Second, we build a trie for $F_0' \cup F_1' \cup \cdots \cup F_{m-1}'$, the union of all FIBs. Note that in this trie, a solid node may have multiple (FIB ID, next hop) pairs. Third, we perform leaf pushing on this trie. Leaf pushing means to push each solid node to some leaf nodes [Srinivasan and Varghese, 1999]. After leaf pushing, every internal node is empty and has two children nodes; furthermore, each leaf node v corresponding to a prefix p is solid and has m (FIB ID, next hop) pairs: $(0, n_0(p)), (1, n_1(p)), \cdots, (m-1, n_{m-1}(p))$, which can be represented as an array H where $H[i] = n_i(p)$ for $0 \leqslant i \leqslant m-1$. Intuitively, we overlay all individual tries constructed from the m FIBs, stretch all tries to have the same structure, and then perform leaf pushing on all tries.

Based on the overlay trie, we run the SAIL_L lookup algorithm. Note that in the resulting data structure, in each next hop array N_{16}, N_{24}, or N_{32}, each element is further an array of size m. Figure 3·6 shows two individual tries and the overlay trie.

Lookup Process: Regarding the IP lookup process for multiple FIBs, given an IP address $\boldsymbol{a}$ and a FIB ID i, we first use SAIL_L to find the next hop array H for $\boldsymbol{a}$. Then, the next hop for IP address $\boldsymbol{a}$ and a FIB ID i is $H[i]$.

Two-dimensional Splitting: Regarding memory management, SAIL_M exactly follows the two-dimensional splitting strategy illustrated in Figure 3·5. We store BC_{16}, which is the combination of B_{16} and C_{16}, and B_{24} in on-chip memory, and store the rest four arrays N_{16}, N_{24}, C_{24}, and N_{32} in off-chip memory. The key feature of our scheme for dealing with multiple FIBs is that the total on-chip memory needed is bounded to $2^{16} * 17 + 2^{24} = 2.13$MB *regardless of the sizes, characteristics and number of FIBs.* The reason that we

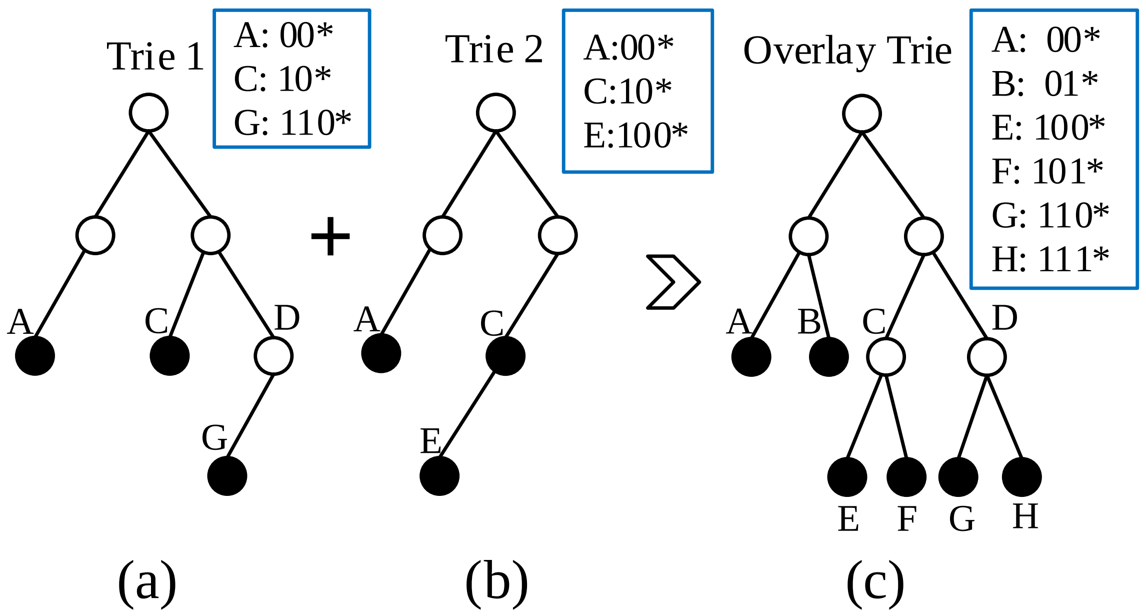


Figure 3·6: Example of SAIL for multiple FIBs.

store BC_{16} and B_{24} in on-chip memory is that given an IP address $\boldsymbol{a}$, BC_{16} and B_{24} can tell us on which exact level, 16, 24, or 32 that we can find the longest matching prefix for $\boldsymbol{a}$. If it is at level 16 or 24, then we need 1 off-chip memory access as we only need to access N_{16} or N_{24}. If it is at level 32, then we need 2 off-chip memory access as we need to access C_{24} and N_{32}. Thus, the lookup process requires 2 on-chip memory accesses (which are on BC_{16} and B_{24}) and at most 2 off-chip memory accesses.

FIB Update: Given a FIB update of inserting/deleting/modifying a prefix, we first modify the overlay trie that the router maintains on the control plane to make the resulting overlay trie equivalent to the updated FIBs. Second, we modify the lookup data structures in the data plane to reflect the change of the overlay trie.

3.4.4 Theoretical Bounds for SAILs

SAIL achieves constant yet small on-chip memory as shown in Table 3.2. For SAIL_B, the upper bound of on-chip memory usage is $(\sum_{i=0}^{24} 2^i)/1024/1024/8 = 4\text{MB}$. For SAIL_U, the upper bound of on-chip memory usage is $(2^6 + 2^{12} + 2^{18} + 2^{24})/1024/1024/8 \approx 2.03\text{MB}$.

Table 3.2: Theoretical Bounds for SAILs.

	On-chip memory	# on-chip mem acc per lookup	# on-chip mem acc per update
SAIL_B	= 4MB	25	1
SAIL_U	$\leqslant$2.03MB	4	1
SAIL_L	$\leqslant$2.13MB	2	Unbounded
SAIL_M	$\leqslant$2.13MB	2	Unbounded

For SAIL_L and SAIL_M, the upper bound of on-chip memory usage is $(2^{16}+2^{16}*16+2^{24})/1024/1024/8 \approx 2.13$MB as we have a bitmap at level 16, and we also have a Chunk ID array at level 16. For on-chip memory accesses, the worst case of all SAIL algorithms ranges from 2 to 25. For off-chip memory accesses, the worst case of all SAIL algorithms is bounded to 2. For SAIL_M, the upper bounds for memory usage and the number of memory accesses are independent of the number of FIBs and the size of each FIB. For SAIL_B and SAIL_U, they only need 1 on-chip memory access per update. However, the update complexity of the other two algorithms is unbounded. In summary, as long as a routers has $\geqslant 4\ MB$ on-chip memory and the off-chip memory is large enough ($\geqslant 4GB$), the lookup speed of SAILs will be fast and bounded. No existing algorithms other than SAIL can achieve this.

3.4.5 Comparison with Typical IP Lookup Algorithms

We qualitatively compare our algorithm SAIL_L with six well known algorithms in Table 3.3 in terms of time and space. Among all these algorithms, only our algorithm has the on-chip memory bound, supports four platforms, and supports virtual FIB lookup. Our algorithm is much faster than all the other algorithms as we will demonstrate in §3.5. In the worst case, our algorithm only needs two off-chip memory accesses. All of these benefits are attributable to our two dimensional splitting and pivot pushing techniques. Our SAIL_L algorithm has the following two shortcomings. The first shortcoming of our SAIL_L al-

Table 3.3: Comparison with Typical IP Lookup Algorithms.

Algorithms	Bounded on-chip memory usage	average lookup speed	worst case #off-chip mem. acc.	platform	virtual FIBs
SAIL_L	✓	around 700Mpps	2	CPU, GPU, FPGA, many-core	✓
PBF	✗	–	25	FPGA	✗
Lulea	✗	around 100Mpps	12	CPU	✗
D18R	✗	around 150Mpps	13	CPU	✗
PopTrie$_{18}$	✗	around 430Mpps	4	CPU	✗
TreeBitmap	✗	around 25Mpps	6	FPGA	✗
LC-trie	✗	around 16Mpps	9	CPU	✗

gorithm is that the update speed in the worst case is unbounded. Fortunately, the average update speed is 2 off-chip memory access per update. If one cares about the update performance most, we recommend the SAIL_U algorithm, which achieves O(1) on-chip update time, but is slower than SAIL_L. Our SAIL_L algorithm only supports CPU, GPU, FPGA and many-core platforms, and the second shortcoming is that it currently does not support multi-core platforms, which will be done in the future work.

3.5 Experimental Results

3.5.1 Implementation

We implemented SAIL_L and SAIL_M on CPU platforms because their data structures have fewer levels than SAIL_B and SAIL_U, and thus are suitable for a CPU platform. For our algorithms on CPU platforms, fewer levels means fewer CPU processing steps. Our CPU experiments were carried out on an Intel(R) Core(TM) i7-3520M. It has two cores with four threads, each core works at 2.9 GHz. It has a 64KB L1 code cache, a 64KB L1 data cache, a 512KB L2 cache, and a 4MB L3 cache. The DRAM size of our computer is 8GB. The actual CPU frequency that we observe in our experiments is 2.82GHz.

3.5.2 Experimental Setup

To obtain real FIBs, we used a server to establish a peer relationship with a tier-1 router in China so that the server can receive FIB updates from the tier-1 router but does not announce new prefixes to the tier-1 router; thus, gradually, the server obtained the whole FIB from the tier-1 router [1]. We use *Quagga* to dump the FIB every hour [Quagga, 2018]. We captured real traffic in one of the tier-1 router's interfaces for the first 10 minutes of each hour between October 22nd 08:00 AM 2013 to October 23rd 21:00 PM.

Prefix Hit Analysis. For the FIBs in most backbone routers, the traffic hits the prefixes of length $<=$ 24. For the FIBs in AS border routers, there could be more traffic hitting prefixes of length $>$ 24. As a result, the lookup speed of SAILs could be slower. However, the worst case of lookup speed is still 2 off-chip memory accesses.

In addition, the datasets of routing tables provided by RIPE NCC are archives of the snapshots. we downloaded 18 snapshots from www.ripe.net [RIPE Network Coordination Centre, 2017]. The first six snapshots were collected from *rrc00* is 8:00 AM on January 1st of each year from 2008 to 2013, respectively, and are denoted by $FIB_{2008}, FIB_{2009}, \cdots, FIB_{2013}$. The other twelve snapshots were taken at 08:00 AM on August 8 in 2013 from 12 different collectors, and are denoted by $rrc00, rrc01$, $rrc03, \cdots rrc07$, $rrc10, \cdots, rrc15$. We also generated 37 synthetic traffic traces. The first 25 traces contain packets with uniformly random destinations from the IPv4 address space [0, $2^{32} - 1$]. The other 12 traces were obtained by generating traffic evenly for each prefix in the 12 snapshots from the 12 collectors at 08:00 AM on August 8 in 2013. We call such traffic *prefix-based synthetic traffic*. The prefix-based traffic is produced as follows: we generally guarantee that each prefix is matched with equal probability. In practice, however, when we generate traffic for a prefix, the produced traffic could match longer prefixes because of the LPM rule. We do not change the order of synthetic traffic. Specifically, given two

[1] It is hardly feasible to dump the FIB of a tier-1 router to hard disk, as the overhead incurred on the router is expensive and we are not permitted to do that due to the administrative policy.

Table 3.4: The percentage of long prefixes of 23 IPv4 FIBs.

FIBs	Total Prefixes	Long Prefixes	Percentage of Long Prefixes
rrc00-2002	114069	2741	2.40%
rrc00-2003	121562	1709	1.41%
rrc00-2004	133214	1376	1.03%
rrc00-2005	165786	9768	5.89%
rrc00-2006	180150	1800	1.00%
rrc00-2007	210072	2564	1.22%
rrc00-2008	245393	4117	1.68%
rrc00-2009	291967	6293	2.16%
rrc00-2010	322152	6562	2.04%
rrc00-2011	348804	3780	1.08%
rrc00-2012	414300	5453	1.32%
rrc00-2013	495401	12178	2.46%
rrc01-2013	460380	1576	0.34%
rrc03-2013	476281	8492	1.78%
rrc04-2013	470772	5495	1.17%
rrc05-2013	467315	1179	0.25%
rrc06-2013	467915	1561	0.33%
rrc07-2013	465539	7034	1.51%
rrc10-2013	467723	6718	1.44%
rrc11-2013	472589	7357	1.56%
rrc12-2013	191601	2440	1.27%
rrc13-2013	373475	28216	7.55%
rrc14-2013	408900	4296	1.05%
rrc15-2013	371877	3094	0.83%

prefixes a_1, a_2, suppose a_2 is after a_1 in the FIB, then the synthetic traffic for a_2 is also after those for a_1. We make the source code for synthetic traffic publicly available at [Yang, 2014].

As the website of ripe.net which is actually the largest open website for FIBs has FIBs from only these locations, we conducted experiments on all available FIBs. We have conducted experiments to test the percentage of longer prefixes, and the results are shown in Table 3.4. This table shows that the percentages of long prefixes in FIBs from different locations and different years are similar, around 1% to 2%.

We evaluated our algorithms on four metrics: *lookup speed* in terms of pps (# of packets per second), *on-chip memory size* in terms of MB, *lookup latency* in terms of microsecond, and *update speed* in terms of the total number of memory accesses per update. For on-chip memory sizes, we are only able to evaluate the FPGA implementation of SAIL algorithms. For lookup latency, we evaluated our GPU implementation because the batch processing incurs more latency.

We compared our algorithms with six well-known IP lookup algorithms: PBF [Dharmapurikar et al., 2003], LC-trie [Nilsson and Karlsson, 1999], Tree Bitmap [Eatherton et al., 2004], Lulea [Degermark et al., 1997], DXR [Zec et al., 2012], and Poptrie [Asai and Ohara, 2015]. For DXR, we implemented its fastest version – D18R. We validated the correctness of all algorithms through exhaustive search: we first construct an exhaustive $2^{32} = 4G$ lookup table where the next hop of an IP address $\boldsymbol{a}$ is the $\boldsymbol{a}$-th entry in this table; second, for each IP address, we compare the lookup result with the result of this exhaustive table lookup, and all our implementations pass this validation.

3.5.3 Performance on CPU

We evaluated the performance of our algorithm on CPU platforms in comparison with LC-trie, Tree Bitmap, Lulea, DXR's fastest version – D18R, and Poptrie. The CPU has two cores, but we only use one core to perform the lookups for all algorithms. We exclude PBF because it is not suitable for CPU implementation due to the many hashing operations.

Our experimental results show that SAIL_L is several times faster than LC-trie, Tree Bitmap, Lulea, D18R, and Poptrie. For real traffic, SAIL_L achieves a lookup speed of 673.22$\sim$708.71 Mpps, which is 34.53$\sim$58.88, 29.56$\sim$31.44, 6.62$\sim$7.66, 4.36$\sim$5.06, 1.59$\sim$1.72 times faster than LC-trie, Tree Bitmap, Lulea, D18R, and Poptrie, respectively. For prefix-based synthetic traffic, SAIL_L achieves a lookup speed of 589.08$\sim$624.65 Mpps, which is 56.58$\sim$68.46, 26.68$\sim$23.79, 7.61$\sim$7.27, 4.69$\sim$5.04, 1.23$\sim$1.32 times

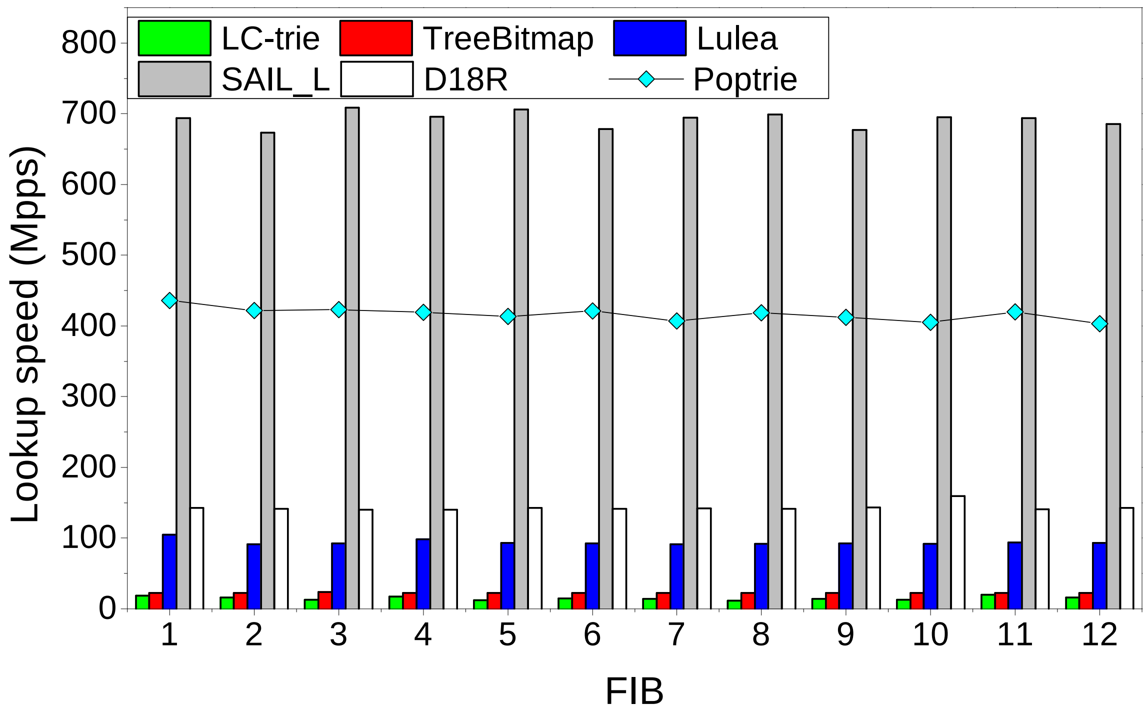


Figure 3·7: Lookup speed with real traffic and real FIBs.

faster than LC-trie, Tree Bitmap, Lulea, D18R, and Poptrie, respectively. For random traffic, SAIL_L achieves a lookup speed of 231.47$\sim$236.08 Mpps, which is 46.22$\sim$54.86, 6.73$\sim$6.95, 4.24$\sim$4.81, 2.27$\sim$2.31, 1.13$\sim$1.14 times faster than LC-trie, Tree Bitmap, Lulea, D18R, and Poptrie, respectively. Figure 3·7 shows the lookup speed of these 6 algorithms with real traffic on real FIBs. The 12 FIBs are the 12 FIB instances of the same router during the first 12 hours period starting from October 22nd 08:00 AM 2013. For each FIB instance at a particular hour, the real traffic that we experimented with is the 10 minutes of real traffic that we captured during that hour. Figure 3·8 shows the lookup speed of these 6 algorithms with prefix-based synthetic traffic on the 12 FIBs downloaded from `www.ripe.net`. Figure 3·9 shows the lookup speed of these 6 algorithms with uniformly random traffic on FIB_{2013}. From these figures, we further observe that for each of these 6 algorithms, its lookup speed on real traffic is faster than on prefix-based traffic, which is still faster than that on random traffic. This is because real traffic has the best IP local-

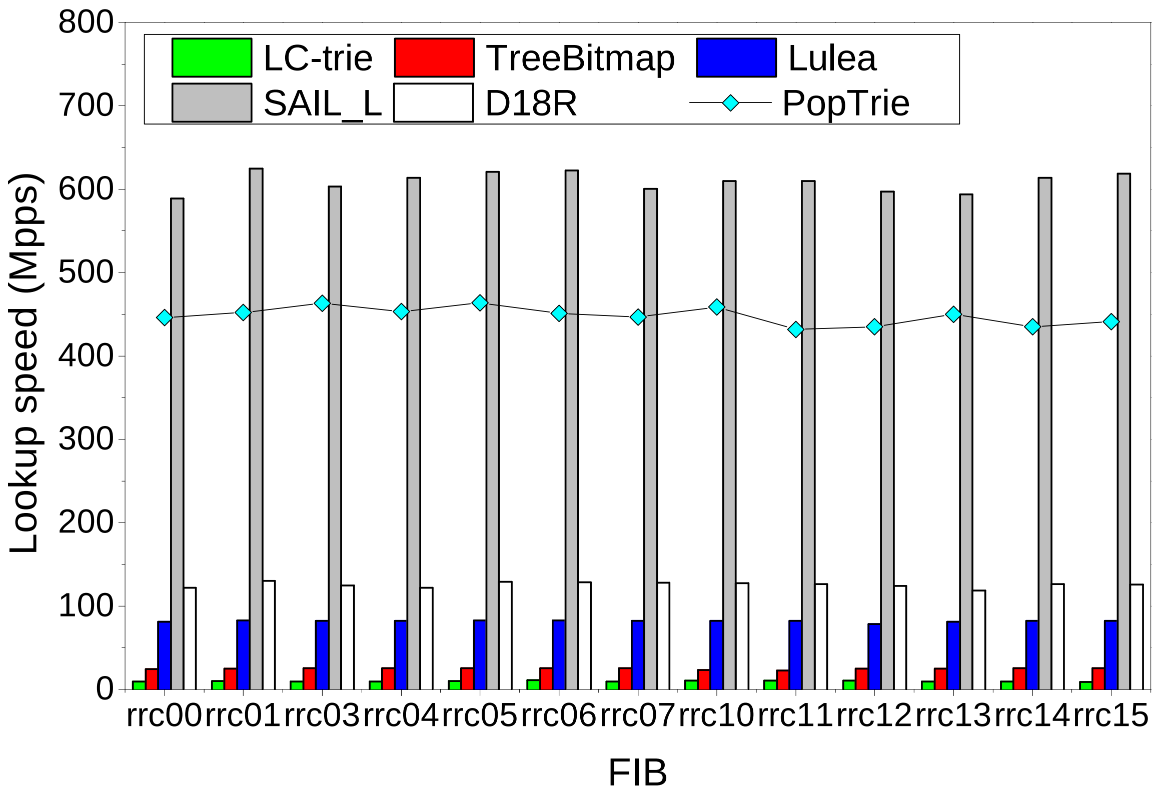


Figure 3·8: Lookup speed with prefix-based traffic on 12 FIBs.

ity, which results in the best CPU caching behavior, and random traffic has the worst IP locality, which results in the worst CPU caching behavior.

As shown in Figure 3·10, the percentages of 25 traffic traces with more than 24 bits matching prefix length range from 0.021% to 0.025%. These percentages are much smaller than the ratio of longer prefixes of the FIB (2.3%), because longer prefixes cover smaller ranges. For example, a leaf prefix node at level 8 covers a range with a width of 2^{32-8}, while a prefix node at level 32 only covers a range with a width of 1. All IP lookup algorithms perform much slower because the cache behaviour is very poor when randomly choosing an IP address from the 4GB space.

We now evaluate the FIB update performance of SAIL_L on the data plane. Figure 3·11 shows the variation of the number of memory accesses per update for 3 FIBs (rrc00, rrc01, and rrc03) during a period with $319 * 500$ updates. The average numbers of memory

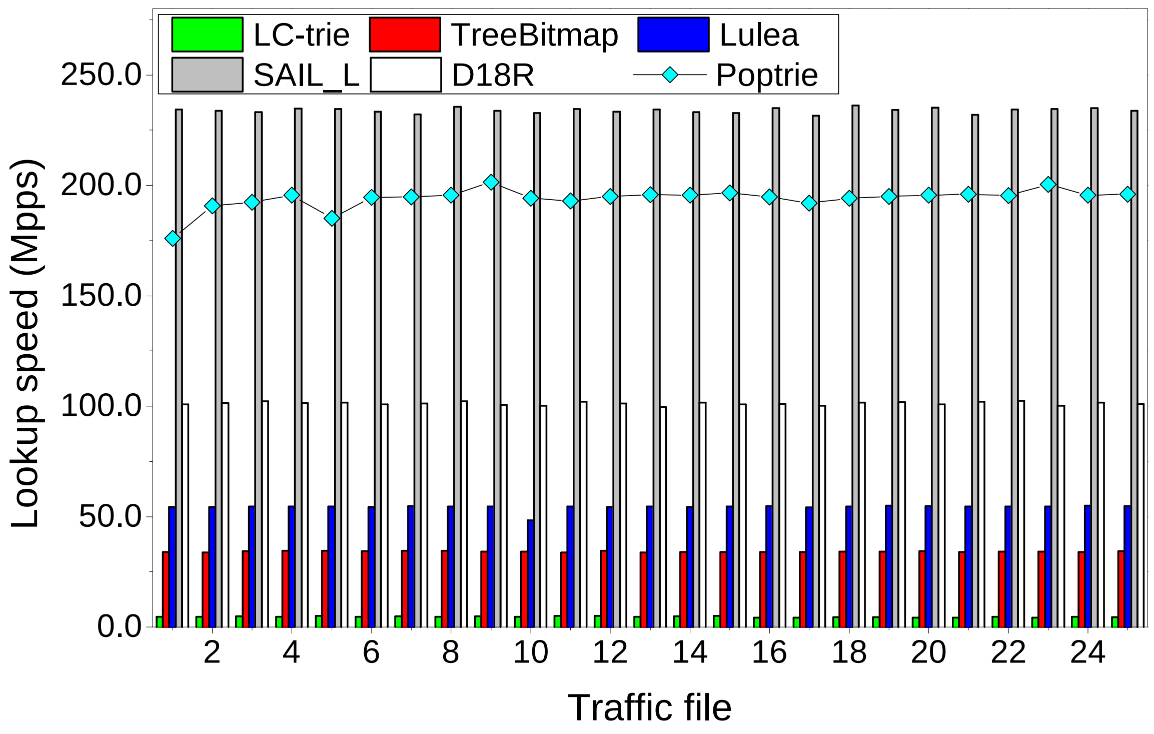


Figure 3·9: Lookup speed with random traffic on FIB_{2013}.

accesses per update for these three FIBs are 1.854, 2.253 and 1.807, respectively. The observed worst case is 7.88 memory accesses per update.

We now evaluate the lookup speed of SAIL_M for virtual routers. Figure 3·12 shows the lookup speed of SAIL_M algorithm as the number of FIBs grows, where x FIBs means the first x FIBs in the 12 FIBs $rrc00, rrc01$, $rrc03, \cdots rrc07$, $rrc10, \cdots, rrc15$, for both prefix-based traffic and random traffic. This figure shows that on average SAIL_M achieves 132 Mpps for random traffic and 366 Mpps for prefix-based traffic.

Lookup speed vs. percentage of long prefixes. As shown in Figure 3·13, the lookup speed of SAIL_L degrades to 448$\sim$467Mpps, 357$\sim$380Mpps, 271$\sim$296Mpps for having $1\%, 5\%, 10\%$ of longer prefixes, respectively. We also compared SAIL_L with Poptrie, which has lookup speeds of 162$\sim$296Mpps, 150$\sim$234Mpps, 124$\sim$186Mpps for having $1\%, 5\%, 10\%$ of longer prefixes, respectively. In conclusion, SAIL_L is 1.47 $\sim$ 2.83 times faster than Poptrie. The performance degradation of both algorithms is mainly because

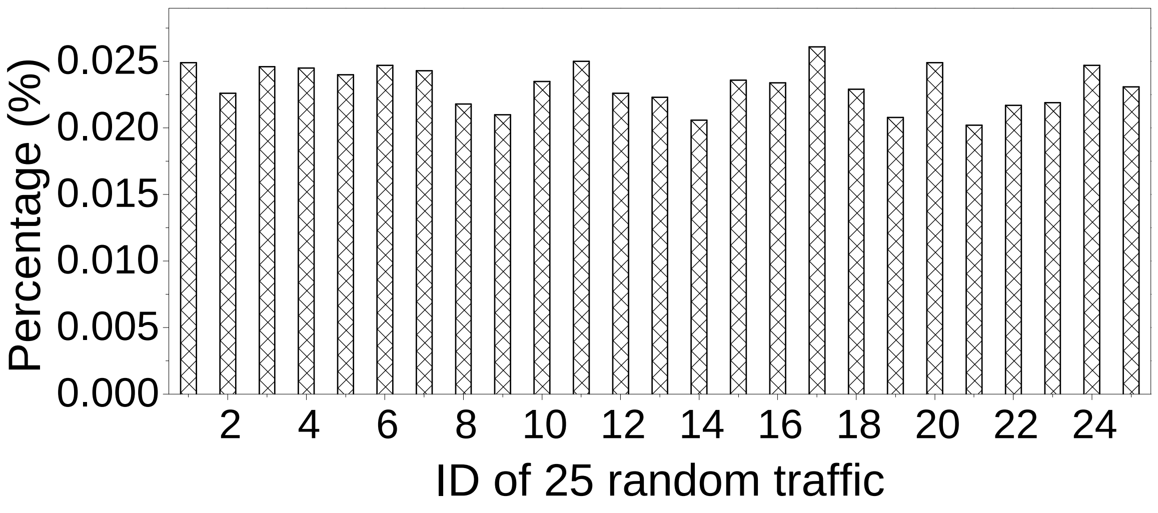


Figure 3·10: Percentages of 25 traffic traces with more than 24 bits matching prefix length for the random traffic.

all the longer prefixes are generated randomly, and the resulted lookup tables have more chunks to maintain comparing with the real FIBs that have better locality.

We exploit technological improvements that the authors of the other algorithms did not have available [Xie et al., 2018a]. Basically, the amount of on-chip memory we have is vastly superior to what there was 10 years ago [Xie et al., 2017, Xie et al., 2018b], and already larger than the theoretical maximum value (4MB) that SAIL needs. So we can implement our lookup algorithm in a much more straightforward way than the others could. So we basically avoid some of the complexities they had to put in to fit whatever was available at the time.

The key idea of Poptrie [Asai and Ohara, 2015] is similar to Lulea [Degermark et al., 1997] as they both compress the next hop arrays based on bitmaps to achieve high memory efficiency. In Poptrie, it uses an SIMD (Single Instruction Multiple Data) instruction named Population Count (*i.e.*, *POPCNT*) which can count the number of 1s in a bitmap. As *POPCNT* is optimized by CPU internal logics, it is much faster than software solutions for counting 1s. Lulea was proposed in 1997 when CPUs did not support *POPCNT*. Thus, Poptrie achieves faster speed than Lulea. Poptrie has four weaknesses when compared with

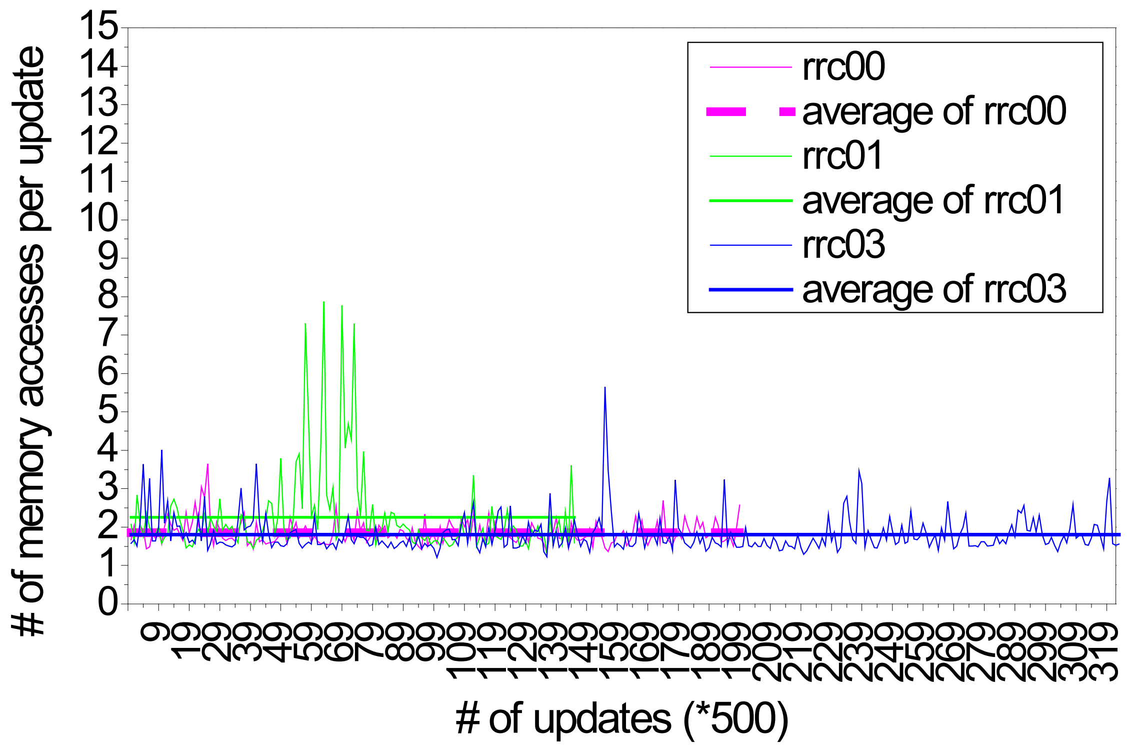


Figure 3·11: # memory accesses per update.

SAIL. First, Poptrie has no memory upper bound and many more memory accesses in the worst case. However, SAIL_L has an upper bound (*i.e.*, 2.13MB) for on-chip memory usage, and at most 2 off-chip memory accesses per lookup. Second, SAIL is considerably faster than Poptrie. Our experimental results on real and synthetic datasets in Section VI.C show that SAIL_L is 1.47 $\sim$ 2.83 times faster than Poptrie. Note that the Poptrie authors did not use our open-sourced SAIL implementation [Yang, 2014]; instead, they implemented SAIL by themselves without publishing the code, we believe their implementation probably introduces some unnecessary overhead that could potentially degrade the performance of SAIL algorithm. And we use their open-source codes to conduct experimental comparison. Third, Poptrie was only implemented in CPU platform, and cannot be implemented on other platforms that do not support *POPCNT*, such as Many-core platform Tirela [Tilera, 2014] and FPGA [Xilinx, 2018] platform. Fourth, the query speed of

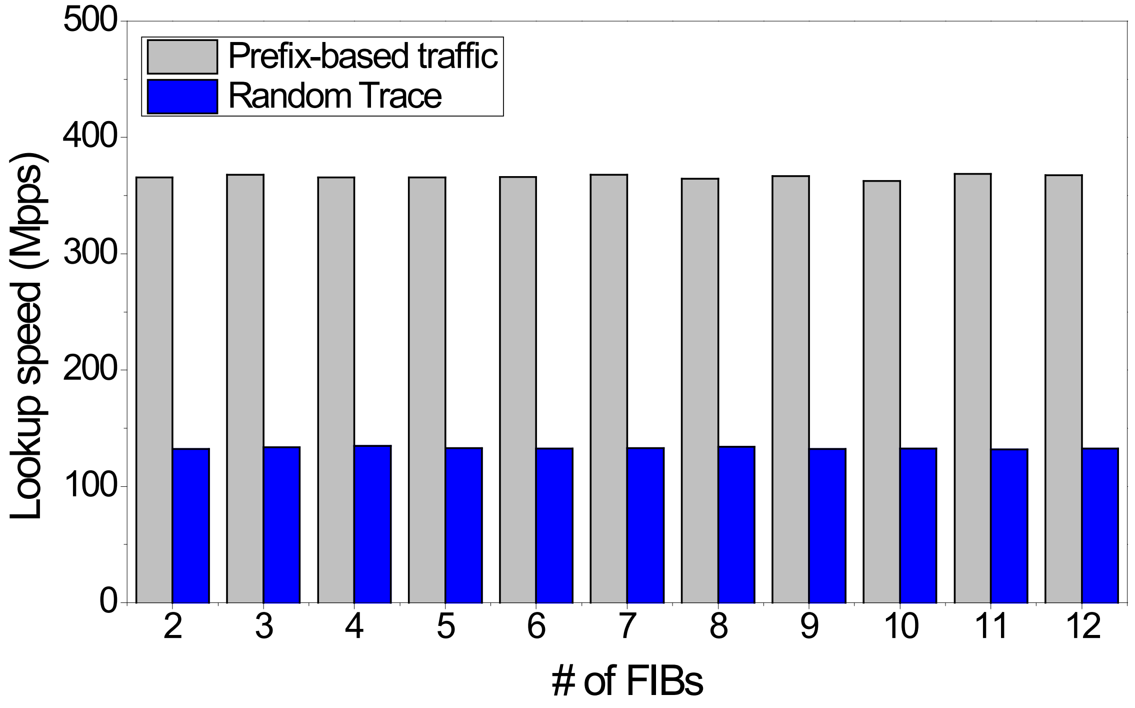


Figure 3·12: Lookup speed of SAIL_M for 12 FIBs using prefix-based and random traffic.

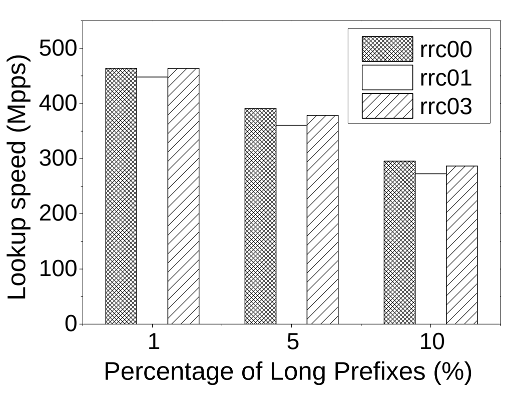
POPCNT will degrade linearly as the number of virtual FIBs increases.

Moreover, the Poptrie paper claims that a previous version of SAIL cannot compile due to its structural limitation. In our previous paper, in level 16, we indeed use 1 bit as flag, and other 15 bits as next hop or ID of chunks. When the FIB is very large, 15 bits for chunk IDs could not be enough. This small problem can be addressed very easily: we can use only 16 bits for next hop or chunk ID, and use additional 2^{16} bits as the flag.

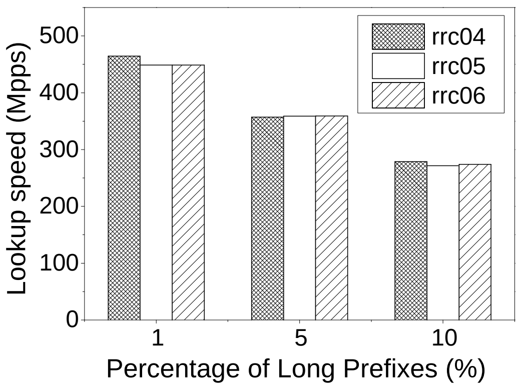
3.6 Scalability for IPv6

3.6.1 SAIL for Small IPv6 FIBs

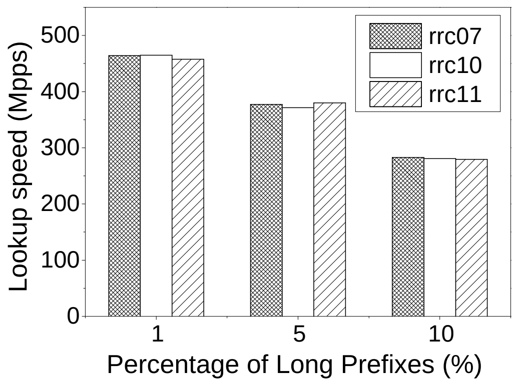
Our SAIL framework is mainly proposed for IPv4 lookup; however, it can be extended for IPv6 lookup as well. An IPv6 address has 128 bits, the first 64 bits represent the network address and the rest 64 bits represent the host address. An IPv6 prefix has 64 bits. The real IPv6 FIBs in backbone routers from `www.ripe.net` only have around 14,000 entries,



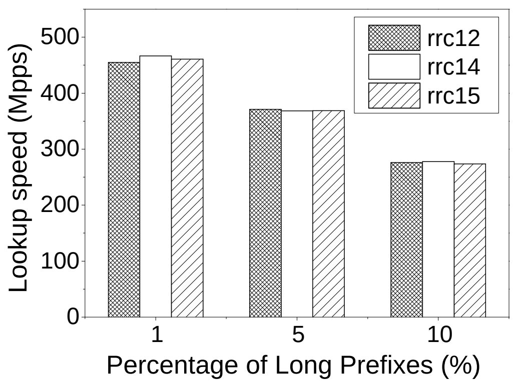
(a) rrc: 00, 01, 03



(b) rrc: 04, 05, 06



(c) rrc: 07, 10, 11



(d) rrc: 12, 14, 15

Figure 3·13: Lookup speed with different percentage of long prefixes on 12 FIBs.

which are much smaller than IPv4 FIBs. To deal with IPv6 FIBs, we can push trie nodes to 6 levels of 16, 24, 32, 40, 48, and 64. To split along the dimension of prefix lengths, we perform the splitting at level 48. In other words, we store the and chunk ID arrays of levels 16, 24, 32, 40, and the array of level 48 in on-chip memory. Our experimental results show that the on-chip memory for an IPv6 FIB is about 2.2MB. Although the on-chip memory usage for IPv6 is much larger than that for IPv4 in the worst case (because IPv6 prefix length are much longer than IPv4), as IPv6 FIB sizes are currently orders of magnitude smaller than IPv4 FIB sizes, the one-chip memory usage for IPv6 is similar to that for IPv4 in today's Internet.

3.6.2 SAIL for Large IPv6 FIBs

For large IPv6 FIBs, since IPv6 addresses are much longer than IPv4 addresses, we can combine the methods of bitmaps in SAIL_U and Bloom filters. Specifically, we can split the prefix length into several parts. Here we give an example (see Figure 3·14): length $0\sim24$, length $25\sim34$, length $35\sim44$, length $45\sim54$, and length $55\sim64$. We propose a *hybrid* scheme that uses bitmaps for the prefixes with length $0\sim24$ and Bloom filters (BF) for others. Bloom filters have false positives whereas bitmaps do not have them. The worst case of using a Bloom filter (PBF [Dharmapurikar et al., 2003]) is when all Bloom filters report true. Although this worst case happens with an extremely small probability, we need to check all the hash tables at level $25 \sim 64$, which is time-consuming.

Pivot inheritance: To reduce the overhead in the worst case, we propose a novel scheme named *Pivot inheritance*. Pivot inheritance is similar to our pivot pushing scheme. First, similar to pivot pushing, we choose several pivot levels. Second, for each pivot level, we only focus on the *Internal and Empty nodes (IE nodes)*. For each IE node, we set its next hop to the next hop of its nearest ancestor solid node. After the IE nodes inherit next hops, we call them *pivot nodes*. Then we show how pivot inheritance alleviates the worst case. Basically, after using pivot inheritance, we insert those pivot nodes in the corresponding hash table, but do not insert them in the Bloom filters.

Example: For example, as shown in Figure 3·14, we select these pivot levels: 24, 34, 44, and 54, and then carry pivot inheritance at these pivot levels. Then, we insert pivot nodes (their prefixes and next hops) into the corresponding hash tables at the pivot levels. Given an incoming IP address $\boldsymbol{a}$, when $\boldsymbol{a}$ matches an internal node at level 24 and all Bloom filters report true, we have the worst case. In this case, our lookup scheme proceeds in the following steps (see Figure 3·15):

Step I: We search $\boldsymbol{a}$ in the hash table at level 44: if the hash table reports a miss, then go to step II; if the hash table reports a leaf node, then the algorithm ends; if the hash table

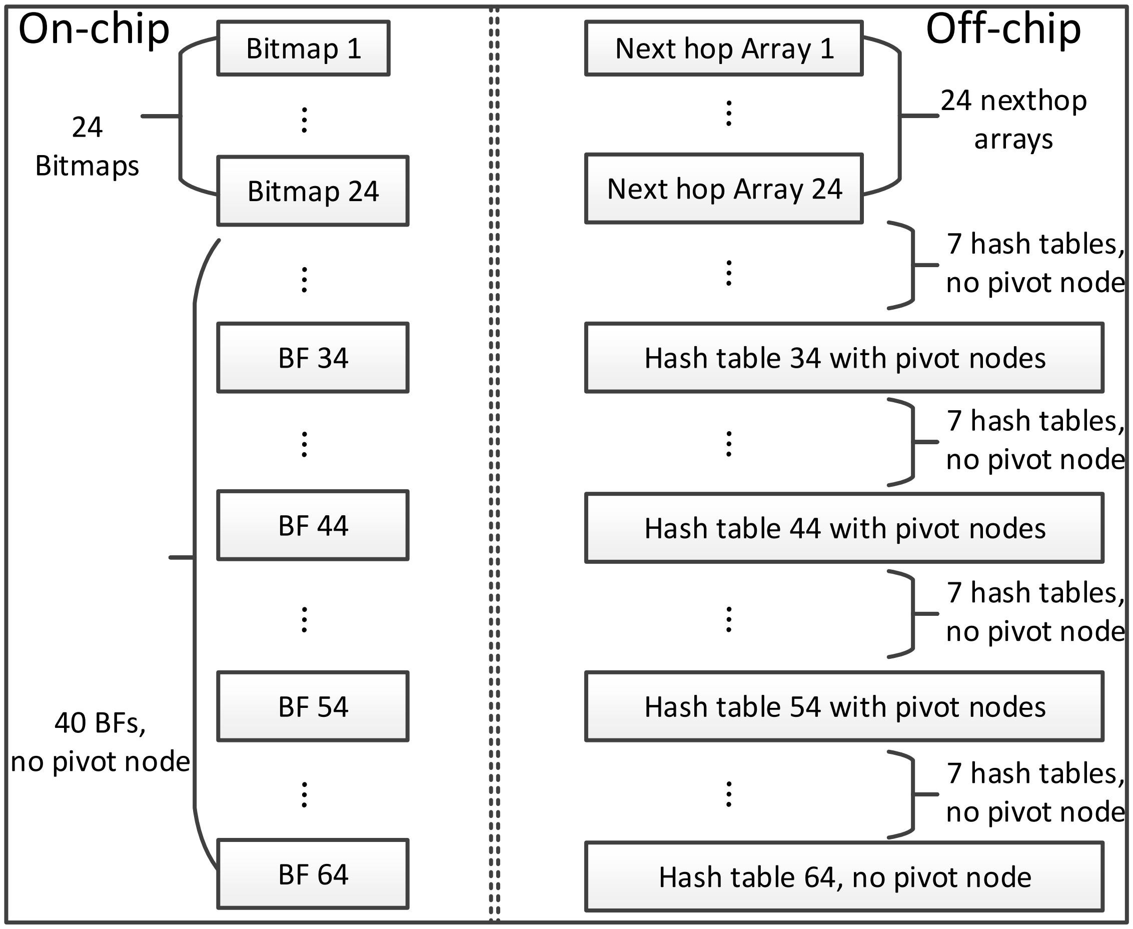

On-chip
Off-chip
Bitmap 1
Bitmap 24
24 Bitmaps
BF 34
BF 44
BF 54
BF 64
40 BFs, no pivot node
Next hop Array 1
Next hop Array 24
24 nexthop arrays
7 hash tables, no pivot node
Hash table 34 with pivot nodes
Hash table 44 with pivot nodes
Hash table 54 with pivot nodes
Hash table 64, no pivot node

Figure 3·14: A hybrid architecture for large IPv6 FIBs.

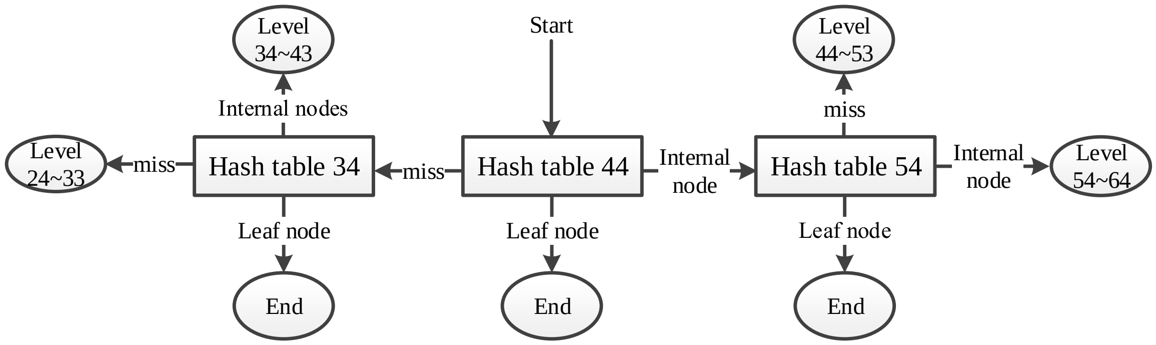


Figure 3·15: Lookup process in the worst case.

reports an internal node, then go to step III;

Step II: We search ***a*** in the hash table at level 34: if the hash table reports a miss, then we know that the matched level is in 24∼33; if the hash table reports a leaf node, then the algorithm ends; if the hash table reports an internal node, then we know that the matched level is in 34∼43;

Step III: We search ***a*** in the hash table at level 54: if the hash table reports a miss, then we know the matched level is in 44∼53; if the hash table reports a leaf node, then the algorithm ends; if the hash table reports an internal node, then we know the matched level is in 54∼64.

Thus, the worst case of the number of hash table searches can be reduced from 40 to 12. More pivot levels can further reduce the number of hash table searches at the cost of more off-chip memory usage. The overhead of our method is reasonably more off-chip memory usage, and we will show it in the experimental results. We need a flag with 2 bits for each node to indicate whether the node is a leaf node, or an internal node, or a pivot node.

3.6.3 Experimental Results for Large IPv6 FIBs

Table 3.5: On-chip memory usage

FIBs	#prefixes (K)	BF memory (MB)	total memory (MB)	free memory (%)
rrc00	455.9	1.04	5.04	24%
rrc01	422.6	0.97	4.97	25%
rrc03	437.9	1.00	5.00	24%
rrc04	432.5	0.99	4.99	24%
rrc05	429.5	0.98	4.98	24%
rrc06	430	0.98	4.98	24%
rrc07	428	0.98	4.98	25%
rrc10	429.6	0.98	4.98	24%
rrc11	434.6	0.99	4.99	24%
rrc12	175.8	0.40	4.40	33%
rrc13	342.7	0.78	4.78	28%
rrc14	375.7	0.86	4.86	26%
rrc15	340.7	0.78	4.78	28%

Experimental Setup

The real-world IPv6 FIBs are small and only have around 13K entries. To evaluate the performance of our algorithm for large IPv6 FIBs, we synthesize IPv6 FIBs based on the real IPv4 FIBs downloaded from www.ripe.net [RIPE Network Coordination Centre, 2017] using the synthetic method proposed in [Wang et al., 2004]. The sizes of synthetic large IPv6 FIBs are similar to those of IPv4 FIBs. The IPv6 FIB sizes are around 10K, while the IPv4 FIB sizes are around 660K in 2017, 66 times larger. The IPv4 FIBs increase around 15% every year. Suppose IPv6 FIBs also increase 15% every year. Then it will take 30 years for IPv6 FIBs to reach the current size of IPv4 FIBs. We conduct experiments for large IPv6 FIBs (see Table 3.5), and results show that our algorithm works well. This means that our algorithm have the potential to work well for next 30 years without upgrading hardware.

On-chip Memory Usage

The on-chip memory usage of our scheme includes two parts: 24 bitmaps and 40 BFs. The maximum memory usage of 24 bitmaps is 4MB. We evaluate the memory usage of 40 BFs using parameters: $k = 13$ and $m/n = 18.755$. The experimental results are shown in Table 3.5, when using synthetic large IPv6 tables. We generate one IPv6 table according to one IPv4 FIB. The sizes of synthetic IPv6 FIBs are similar to the corresponding IPv4 FIBs. For example, the size of IPv4 table rrc00 is 455,900, and the size of the synthetic IPv6 table is also 455,900. The number of prefixes at level 25$\sim$64 is around 400K, and the BF memory usage is around 1MB. Thus, the total memory is around 5MB, while the on-chip memory capacity of old-fashioned FPGA is 6.6MB [Xilinx, 2018]. Modern ASICs have 50$\sim$100MB on-chip memory [Miao et al., 2017].

Additional Off-chip Memory Usage

With pivot inheritance, the worst case of this hybrid scheme can be reduced to 12 hash table searches when using 4 pivot levels. The overhead of pivot inheritance is the additional off-chip memory usage.

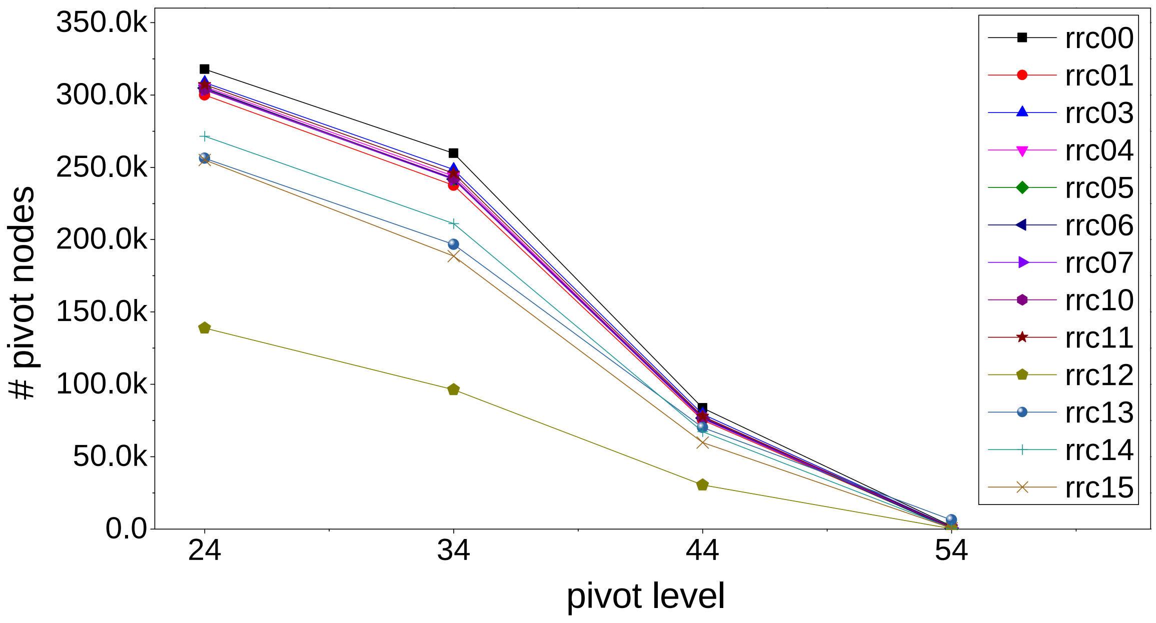


Figure 3·16: The number of pivot nodes at the pivot levels.

We carry out pivot inheritance using the large synthetic IPv6 FIBs based on the IPv4 FIBs downloaded from [RIPE Network Coordination Centre, 2017]. We choose the four levels (24, 34, 44, and 54) as the pivot levels. In practice, for IPv4 FIBs, users can flexibly choose appropriate pivot levels. Here we only show an example for synthetic FIBs. It can be observed from Figure 3·16 that the number of pivot nodes is around 300K at level 24, but only 1K at level 54. The total number of pivot nodes is a little larger than the size of synthetic IPv6 FIBs. Fortunately, these pivot nodes only consume additional off-chip memory, but do not consume on-chip memory at all. We show that the additional and total memory usage using pivot inheritance in Figure 3·17. It also shows the memory usage of Tree Bitmap on large synthetic FIBs. The source code of Tree Bitmap is available

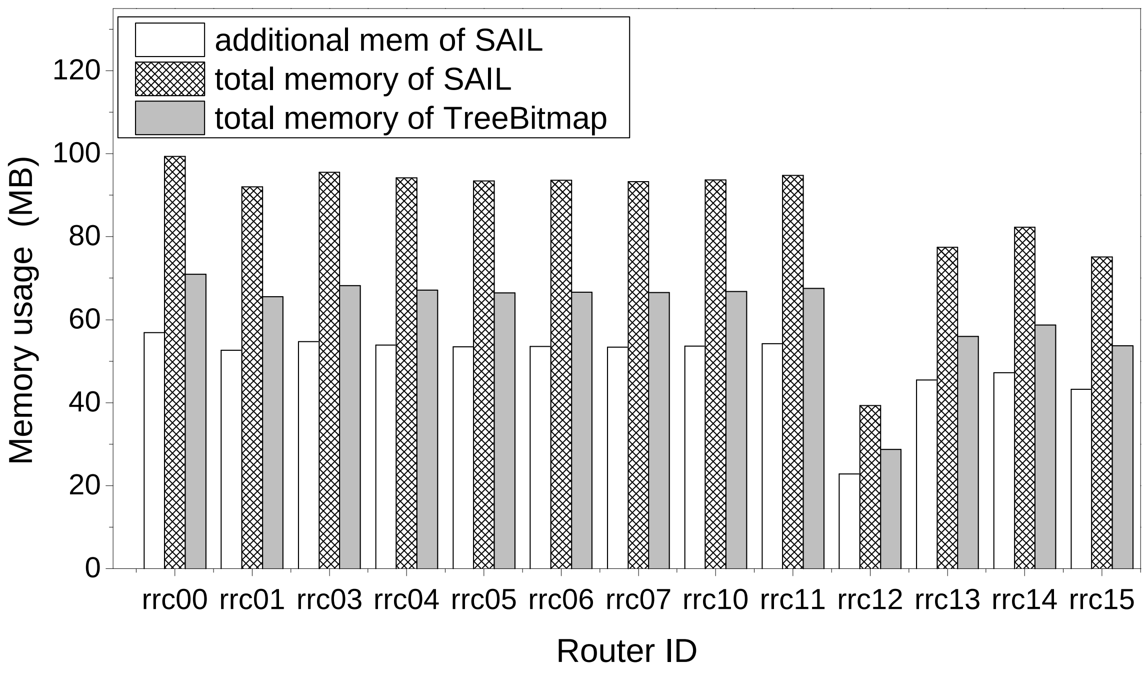


Figure 3·17: The additional and overall off-chip memory usage.

at [Yang, 2014]. For the sake of simplicity, we assume that all off-chip data structures use hash tables. To achieve small probability of hash collisions, we set the load factor to be 0.1. This means that for n elements, we use $10 \times n$ hash buckets. Using these parameter settings, the experimental results are shown in Figure 3·17. It can be seen that the additional memory usage ranges from 22.8 MB to 56.9 MB with a mean of 49.6 MB, while the total memory usage ranges from 39.3 MB to 99.3 MB with a mean of 86.5 MB. Nowadays, the off-chip memory capacity is about 100 times of the above total memory usage of the hash tables. Since the off-chip memory is cheap, the overhead of pivot inheritance is reasonable and affordable. Experimental results also show that the total memory usage of Tree Bitmap ranges from 28.7 MB to 71.1 MB with a mean of 61.8, and it is about 71.4 % of that of our SAIL algorithm. However, Tree Bitmap does not split the memory into on-chip and off-chip memory, and one lookup needs multiple off-chip memory accesses. In contrast, we only need less than 5 MB on-chip memory usage, and $10\times$ buckets in the off-chip hash tables to achieve the speed of about 1 off-chip memory access per lookup.

Analysis of IPv6 lookup performance: In the best case, only one Bloom filter reports yes, then the lookup needs one hash table probe, *i.e.*, one memory access [Dharmapurikar et al., 2003]. In the worst case, the longest matched prefix is at level 24 or 54, and all Bloom filters report yes. Then, our algorithm needs $2+10$ hash table probes. As we use 13 hash functions for each Bloom filter, the false positive rate is $0.5^{13} = 2^{-13}$ when using the optimal setting. The probability that the worst case happens is $0.5^{13*(10-1)} = 2^{-117}$. On average, each lookup needs around one hash table probe.

3.7 Summary

We make four key contributions in this chapter. First, we propose a two-dimensional splitting approach to IP lookup. The key benefit of such splitting is that we can solve the sub-problem of finding the prefix length $\leqslant 24$ in on-chip memory of bounded small size, with the help of our proposed *pivot pushing* algorithm. Second, we propose a suite of algorithms for IP lookup based on our SAIL framework. One key feature of our algorithms is that we achieve small constant IP lookup time and on-chip memory usage. Another key feature is that our algorithms are cross-platform as the data structures are all arrays and only require four operations of ADD, SUBTRACTION, SHIFT, and logical AND. Note that SAIL is a general framework where different solutions to the sub-problems can be adopted. The algorithms proposed in this chapter represent particular instantiations of our SAIL framework. Third, we extend our SAIL approach to IPv6, and proposed a hybrid scheme. To improve the performance of synthetic large IPv6 FIBs in the worst case, we propose a novel technique called *pivot inheriting*. Fourth, our algorithms support four platforms (namely FPGA, CPU, GPU, and many-core) and we conducted extensive experiments to evaluate our algorithms using real FIBs and traffic from a major ISP in China. Our experimental results show that our SAIL algorithms are several times or even two orders of magnitude faster than the well known IP lookup algorithms. Furthermore, we have open sourced our SAIL_L algorithm and four well known IP lookup algorithms (namely LC-trie, Tree Bitmap, Lulea, and DXR) that we implemented.

Chapter 4

Gatekeeper: First Open-source and Deployable DDoS Mitigation System

This chapter introduces Gatekeeper - the first open-source and deployable DoS mitigation system. Specifically, Gatekeeper is an architectural approach to mitigating DoS attacks that is instantly deployable: any autonomous system (AS) can reap the full benefits of Gatekeeper with its own deployment. Gatekeeper leverages the existing infrastructure of geographically-distributed and well-provisioned vantage points (*e.g.*, Internet exchanges and some cloud providers) outside of the deploying AS. Governed by centralized policies managed by the deploying AS, vantage points can drop attack traffic close to the source without the coordination of other network operators. Since packet processing is on the critical path in Gatekeeper, we have to optimize it as much as possible to achieve high performance even in the worst case (*i.e.*, every packet is a request). In the first deployment, our goal is to achieve 10 Gbps throughput on a single commodity server. This chapter describes several performance optimization techniques over a baseline implementation to achieve 10 Gbps throughput. Especially, we rely on batched lookup and memory prefetching. Moreover, as the ultimate goal of Gatekeeper is to achieve 100+ Gbps throughput on a single commodity server, we explore several hash table algorithms that will help redesign the flow hash table in Gatekeeper to further improve its performance.

4.1 Introduction

According to Arbor Networks [Anstee et al., 2017], DoS attack has been the top operational concern among their surveyed entities, including enterprise, government and education respondents. Since the first 10Gbps DDoS attack in 2005, the peak capacity of DDoS attacks is more than doubling every two years. In 2018, DDoS attack size exploded to a record-breaking 1.7Tbps in size [NETSCOUT, 2019], which could have a devastating impact on most of the targeted victims. Moreover, the number of attacks is more than doubling every year. Fueled by the advancement of Internet of Things (IoT) and cloud computing, DoS attacks could have even-higher attack rates [Antonakakis et al., 2017, NETSCOUT, 2019, Miao et al., 2015].

4.2 Related Work

To defend against DoS attacks, many architectural solutions including [Ferguson and Senie, 2000, Mahajan et al., 2002, Andersen et al., 2003, Anderson et al., 2004, Yaar et al., 2004, Argyraki and Cheriton, 2005, Greenhalgh et al., 2005, Parno et al., 2007, Liu et al., 2008, Dixon et al., 2008, Kang et al., 2013, Fayaz et al., 2015, Basescu et al., 2015, Yang et al., 2005] have been proposed. Among these solutions, there are two classes that attracted much of the researchers' attention: capability-based and filtering-based approaches. Basically, capability-based solutions require that a source node must first request permission to send to a destination node, and destination authorizes source for communication. Authorizations are ultimately substantiated as tokens that packets carry, and routers on the path of communication validate before forwarding packets. These tokens are called network capabilities. In contrast, filtering-based solutions require that destinations grant access by default, and halt access by senders identified as malicious. One can see filtering as a blacklist approach, where the receivers ask the routers on the path to install network filters for misbehaving senders. Among them, several solutions offer an incremental deployment plan, however, their effectiveness depends on the number of autonomous systems (ASes) that deploy it. Since these solutions derive much of their effectiveness from network effects, there is minimal incentive for pioneering ASes to deploy the solution initially, therefore, they risk never overcoming the initial activation energy needed to deploy the system widely.

4.3 Gatekeeper Overview

Gatekeeper is the first open-source DDoS mitigation system designed for a single AS, thus it does not need any collaboration with other ASes. Gatekeeper combines the strength of both capability and filtering systems. Gatekeeper aims to achieve 10 Gbps throughput on a single commodity server for the first deployment, and ultimately achieve 100+ Gbps throughput on a single commodity server. Gatekeeper can be easily scaled to any peak bandwidth with more nodes, thus it can withstand DoS attacks both of today, and of tomorrow. Moreover, Gatekeeper takes advantage of the well-provisioned and geographically distributed vantage points (see §4.3.1) to match the peak capacity of DDoS attacks. As all inbound traffic must go through a vantage point, the attack packets can be dropped close to the source. Gatekeeper centralizes its network policy management in the Grantor servers (see §4.3.3) at the deploying AS. Grantors make policy decisions on all incoming flows and install them to Gatekeeper servers (see §4.3.2) located inside vantage points for policy enforcement. Additionally, Grantors support flexible policies (*e.g.*, blacklist, whitelist, machine learning-based) to fight multiple multi-vector DoS attacks at once. We have a fully functional implementation of Gatekeeper, which complies with industry practices and standards, and is ready for production deployment. The codebase of Gatekeeper is publicly available on GitHub [Machado et al., 2020] with more than 33 thousands lines of code (KLOC) and over 700 commits. To the network community, Gatekeeper is a template to make other architectural solutions deployable by a single AS.

Figure 4·1 gives an overview of Gatekeeper with two highlighted components: Gate-

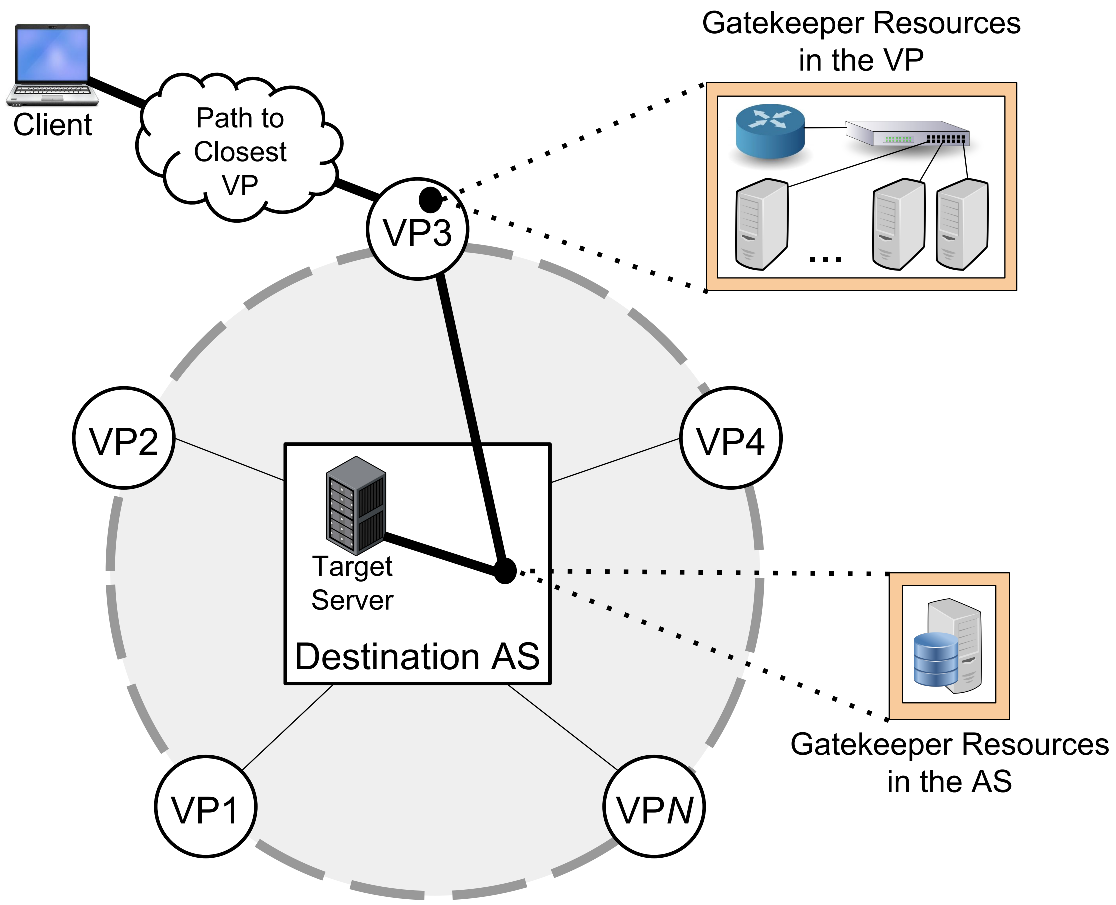


Figure 4·1: An overview of the components of Gatekeeper and the role of vantage points in providing a protective shield around the destination AS.

keeper servers and Grantor servers. Gatekeeper servers are deployed throughout the Internet at locations called Vantage Points (VPs), which provide a protective shield around the deploying AS. Internet eXchange Points (IXPs), some cloud providers, and peering links are examples of vantage points. When a client wants to communicate with a protected destination node, the client must first request permission to send to the destination node from a Grantor server located in the data center of the defended AS. Grantor servers determine the fate of requests (*e.g.*, declined, granted with rate limit, etc.) to the protected servers according to the network policies written by the deploying AS. The policy decisions will be installed as network capabilities at Gatekeeper servers, which enforce them on all the incoming traffic to the destinations protected by Gatekeeper.

4.3.1 Vantage Points

A prospective Gatekeeper vantage point (VP) is any network location that meets four requirements: (1) computing capacity, (2) cheap ingress bandwidth, (3) BGP peering, and (4) private links to the protected AS. (1) and (2) are basic requirements to ensure that the VP can support the placement of machines to affordably run a DoS mitigation service. Some candidate VPs, such as cloud providers like Amazon AWS, Microsoft Azure, and Google Cloud, even provide customers with free inbound data transfers. Requirements (3) and (4) are motivated by the design of Gatekeeper. BGP peering is needed so that an anycast network is created by announcing the same BGP route in multiple VPs, where VPs act as boundaries close to the sources. Finally, (4) is essential because the path between a VP and the deploying AS must not expose routable addresses to the open Internet. Otherwise, these links are vulnerable to DoS attacks themselves.

Internet Exchange Points (IXPs) are vantage points by this definition, and they are ideal locations for deploying Gatekeeper for the following main reasons: (1) IXPs typically provide services to many ASes, so that the member ASes can share the capital expenditure necessary for Gatekeeper deployment; (2) The member ASes can deploy Gatekeeper without cooperation and coordination among them; (3) IXPs have sufficient capacity to absorb multi-Tbps DDoS attacks; (4) IXPs have experienced network management teams to reduce the operational costs necessary for Gatekeeper deployment. An example IXP is DE-CIX, which has more than 800 networks and 25 Tbps connected capacity. Besides, some cloud providers and peering links are also examples of vantage points. Some VPs, such as cloud providers, do not provide BGP peering services, but companies such as Datapath.io [Datapath.io, 2017] can help Gatekeeper deployments to overcome this hurdle.

4.3.2 Gatekeeper Servers

Gatekeeper servers at multiple VPs use BGP to announce the same network prefixes under their protection. Thus, an anycast network is created and each traffic source is bound to a VP. Therefore, all inbound traffic *must* go through a VP. Inside the VP, the router in Figure 4·1 connects to the VP and load balances incoming flows to the Gatekeeper servers using ECMP. Note that a flow is defined as the pair (source, destination) addresses. Each Gatekeeper server then performs the following tasks: (1) bookkeeping capabilities for each flow; (2) limiting bandwidth as specified by the capabilities; and (3) encapsulating packets to send to the Grantor in the destination AS. When a Gatekeeper server does not yet have a policy decision in its flow table to enforce on a given flow, it encapsulates the packet of that flow using IP-in-IP, assigns a priority to the encapsulated packet based on the rate of the given flow (higher priority for lower rates), and forwards it through the request channel. The request channel is reserved 5% of the bandwidth of the path that goes from a Gatekeeper server to the Grantor server responsible for the policy decision. Whenever the request channel exceeds its capacity, packets with the lowest priority will be dropped.

4.3.3 Grantor Servers

As packets sent to Grantor servers are encapsulated using IP-in-IP, Grantor servers need to decapsulate the packets, make decisions on requesting flows, and forward granted flows to their destinations. When processing requests, Grantor checks each request against the network policies and sends the decisions to the corresponding Gatekeeper servers to enforce them. In our implementation, a network policy is a Lua script that runs on Grantor servers. Grantor supports various policies, *e.g.*, blacklists, source address authentication, control-theoretic methods (AIMD), machine learning techniques (per-AS reputation), extended Berkeley Packet Filtering (eBPF) programs, and is designed to be extensible. When processing granted flows, Grantor simply decapsulates the packet and sends it to the destination.

Background on eBPF

The Berkeley Packet Filter (BPF) was originally developed as a component of the kernel networking subsystem on Unix-like operating systems to filter unwanted packets as early as possible [McCanne and Jacobson, 1993], so that it minimized the packet copying from kernel to userspace processes. These filters are implemented as programs to be run on a register-based virtual machine. BPF can be used in user space, although it was written for kernel space. Specifically, it allows user-space programs to securely run inside the kernel. For example, `tcpdump` uses BPF to monitor the network traffic. Many architectures (*e.g.*, x86_64, ARM) support the just-in-time (JIT) compiler to optimize the BPF programs on the fly.

The extended Berkeley Packet Filtering (eBPF) was introduced in 3.15 kernel by Alexei Starovoitov in his patch, which takes advantage of the advancement in modern CPU [Fleming, 2017]. Specifically, this patch increased the number of registers from two in original BPF implementation to ten, and the register width from 32-bit to 64-bit. In this way, the eBPF instructions can be modeled closer to the hardware instruction set architecture (ISA) for improved performance. Moreover, it allows eBPF programs to call a restricted subset of kernel functions cheaply. eBPF also provides several pre-defined data structures (*e.g.*, map, hash, array) to share data within the kernel or between the kernel and user space. Although eBPF is very suitable for writing network programs (*e.g.*, enforcing network classifier actions in Gatekeeper system), nowadays it also has been used in kernel debugging, performance analysis, monitoring, and even user-space program debugging, etc. Notice that eBPF programs can be loaded into the kernel through the `bpf()` system call. Later, the eBPF in-kernel verifier performs several checks on the eBPF programs to make sure that they can be run inside the kernel in a safe manner.

Historically, the developers had to write eBPF programs in low-level language. With the support from compilers, such as LLVM Clang and GCC, the developers can write eBPF programs in C and then compile them into eBPF bytecodes. The BPF Compiler Collection (BCC) project even allows developers to write eBPF programs in Python and Lua [IO Visor Project, 2015]. DPDK provides an BPF library, which allows network applications to load and execute eBPF bytecode in user-space [DPDK Project, 2020].

In Gatekeeper system, policy decisions can be BPF programs, so they can limit or act on changes of behavior. We have implemented several examples of BPF programs. For example, the grantedv2 eBPF program limits traffic with two rate limits: primary and secondary limits or channels. Specifically, all traffic is subject to the primary limit, and the undesirable traffic (*e.g.*, fragmented packets) that fits the primary limit is subject to the second limit. Since Grantors are using Lua to manage their policies, BCC allows Grantors to create BPF programs on-the-fly inside of the Lua policy, which opens the door to more flexible and dynamic policies decisions in Gatekeeper.

4.3.4 Queue management

Gatekeeper nodes – Gatekeeper servers and routers on the VP-AS channel – queue packets according to their type, priority, and destination address (see Figure 4·2). Packet types are recognized according to the 6-bit Differentiated Services Code Point (DSCP) field of IP headers: 0 for operational packets; 1 for granted packets; 2 for granted packets renewing a capability; and 3 – 63 for capability requests.

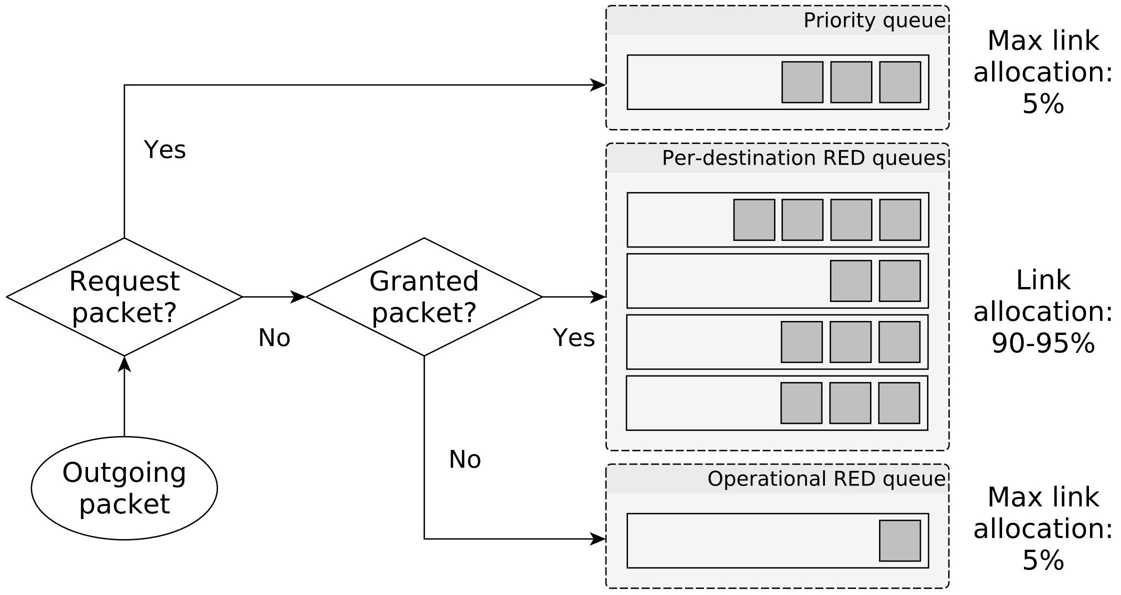


Figure 4·2: Queue management at Gatekeeper servers and at any capable router on the VP-AS path.

Each capability request is placed into a priority queue according to the value of its DSCP field; the higher the priority (*i.e.* a higher value of the DSCP field), the closer to the exit of the queue the request is placed. When the queue is full, the request with the lowest priority is dropped. To help mitigate attacks against the request channel, the request queue receives only 5% of the bandwidth capacity of the outgoing link. Granted packets, that is, packets that the destination has previously authorized, are placed in RED queues [Floyd and Jacobson, 1993]. The number of queues depends on hardware capacity of the node, and a queue is chosen by hashing the destination address. The seed of this hash changes periodically to minimize the effects of hash collisions. The queues used for granted packets command up to 95% of the bandwidth capacity of the outgoing link. To avoid communication disruption between routers when the system is under attack, an operational queue reserves 5% of bandwidth for essential protocols such as ARP, IPv6 ND, ICMP, and BGP.

A critical aspect of our system is the priority allocated to each request packet, which (in addition to the 5% request channel bandwidth limit) helps prevent attacks on the request channel. Since our system is designed to support a single AS, we assume that the Gatekeeper servers are trusted, which allows us to employ a simpler solution than the one used in Portcullis [Parno et al., 2007]. For example, the priority assignment algorithm simply returns the log of the packet's waiting time, *i.e.*, $\log_2(delta\_time)$. Therefore, senders who wait longer to send a packet are assigned a higher priority, which is similar to the exponential back-off method used in Portcullis.

Ideally, all routers between Gatekeepers and destinations use the same queue management tools, but this may not always be possible due to legacy routers. We expect that at least key routers on the VP-AS path enforce the queue management described above. Our queue management is mostly a refinement of what had been proposed in prior work [Yang et al., 2005, Parno et al., 2007]. To manage the request queue in Gatekeeper servers, we implement it as a separate functional block - the Solicitor (SOL) block, which is responsible for maintaining, rate limiting, and sending requests. Request packets are organized by priority and only permitted a fraction of the link capacity between Gatekeeper and Grantor, and the Solicitor block enforces these constraints.

To summarize, Gatekeeper is a combination of several previous approaches in the literature to mitigates attacks. It is a redirection-based architecture in the sense that it redirects packets bound for a target destination through the Gatekeeper system. It is a capability-based architecture in the sense that flows have to get permission from Grantor based on a policy before being allowed to send. And it is a filter-based architecture in the sense that for packets that are to be dropped, filters are installed upstream (at the Gatekeeper server) to prevent wasting downstream resources.

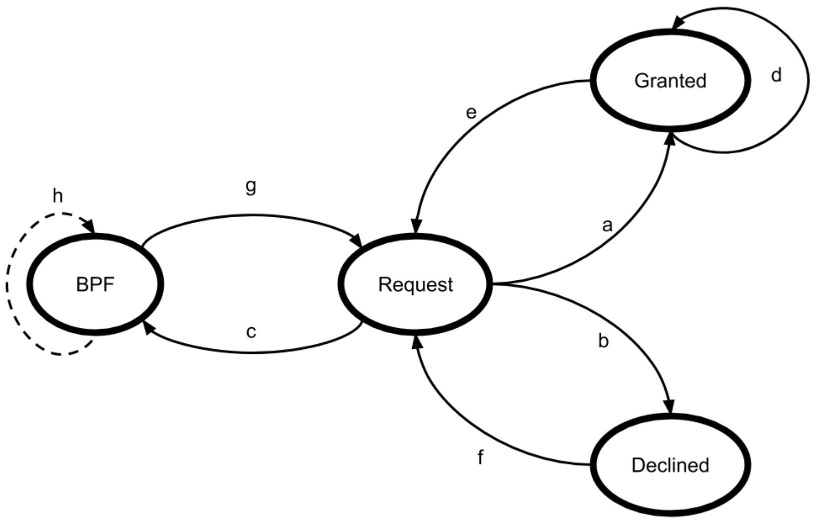


Figure 4·3: Flow entry state transition

4.4 Packet Processing in Gatekeeper

4.4.1 Flow States

In Gatekeeper, each incoming packet will be associated with a flow entry. A flow entry can be in one of four states: Request, Granted, Declined, and BPF. These states and their transitions are depicted in Figure 4·3. In general, when a flow entry is first created, its state is Request. An entry only moves from the Request state to one of the other three states after the Gatekeeper server processes a reply from a Grantor server. These possible transitions are named (a), (b) and (c) in the figure. If a Grantor accepts the request, then its flow entry is at Granted state, and will remain until the state expires. If a Granted state expires, the entry goes back to Request state (e). Gatekeeper will try to renew a Granted state before it expires (d). If a request is declined, then its flow entry is at Declined state, it only leaves this state when it expires (f). If a Grantor instructs the corresponding Gatekeeper server to run a BPF program to enforce policy decisions for the request, then its entry is at BPF state, it only leaves this state when it expires (g). Gatekeeper will optionally try to renew a BPF state before it expires (h). Notice that in BPF state, Gatekeeper will take the flow state associated with the incoming packet as input of the BPF programs installed by Grantor servers, and the BPF programs will determine the fate of the incoming packet. Two currently supported decisions are *forwarding the packet* and *declining the packet*.

4.4.2 Packet Processing Algorithms

Algorithm 5: Basic packet processing algorithm in Gatekeeper servers

```
Input: A packet received from the NIC: pkt
Input: A flow hash table: flow_hash_table
Input: An LPM table: lpm_tbl
1  init flow  /* IP flow placeholder for the packet */
2  init fe    /* flow entry placeholder for the packet */
3  init fib   /* FIB entry placeholder for the packet */
4  flow = extract_packet_info(pkt)
5  fe = hash_table_lookup(flow_hash_table, flow)
6  if fe == null then
7      fib = lpm_fib_lookup(lpm_tbl, flow)
8      fe = lookup_fe_from_lpm(flow, pkt, fib)
9  end
10 if fe != null then
11     process_flow_entry(fe, pkt)
12 end
```

Algorithm 5 shows the pseduocode of the basic packet processing algorithm for the Gatekeeper servers. The capability information is stored in flow entries of the flow hash table. The policy table is an LPM table including policies on destinations. To summarize, when a packet arrives at a Gatekeeper server, the server first extracts the packet's IP flow information (Line 4), and conducts a flow hash table lookup to find the associated flow entry (Line 5). If there is no flow entry associated with the incoming packet in the flow hash table, it looks up the LPM policy table to process the packet accordingly (Lines 6 − 9). We will present how the packet missing the flow hash table lookup is processed by using the FIB entry obtained from the LPM policy table lookup in Algorithm 6. After obtaining the associated flow entry, it processes the packet as the flow entry instructs (Lines 10 − 12), which will be detailed in Algorithm 7. To improve cache locality, instead of storing the flow entries directly in the flow hash table, we store them in a separate array with the same size as the flow hash table. Thus, only the index of the flow entry in the array is stored in the hash table. Moreover, Gatekeeper periodically scans over the flow hash table to remove the expired ones. Notice that as the packets will be either dropped or sent out, the algorithms in this and following sections do not return anything unless explicitly stated.

Algorithm 6: Lookup flow entry using the FIB entry from policy table

```
   Input: A packet: pkt
   Input: The flow information associated with pkt: flow
   Input: FIB information associated with pkt from policy table: fib
   Output: The flow entry in the flow hash table associated with pkt
 1 init fe   /* flow entry placeholder */
 2 switch fib.action do
 3     case FWD_GRANTOR do
 4         fe = add_flow_entry(flow, pkt)
 5         initialize_flow_entry(flow, fe, fib)
 6     end
 7     case FWD_BACK_NET do
 8         forward_to_back_network(pkt)
 9     end
10     otherwise do
11         drop_packet(pkt)
12     end
13 end
14 return fe
```

Algorithm 6 shows the pseudocode of the flow entry lookup algorithm given the FIB entry from the LPM policy table lookup. If the FIB entry from the policy table instructs to forward the packet to a Grantor server, then it inserts a new flow entry into the flow hash table and initializes the flow entry (Lines 3 − 6). Gatekeeper also supports flow bypasses (Lines 7 − 9), which matches flows and forwards them to the back interface accordingly (*e.g.*, forward the packet to either the gateway or a neighbor in the back network). Bypasses

Algorithm 7: Packet processing algorithm based on flow state

```
Input: A packet: pkt
Input: The flow entry associated with the packet: fe
if fe.state == REQUESTING then
    priority = priority_from_delta_time(now, fe)
    encapsulate(fe, pkt, priority)
    gk_solicitor_enqueue(pkt, priority)
else if fe.state == GRANTED then
    if is_valid_capability(fe, pkt) then
        encapsulate(fe, pkt, PRIORITY_GRANTED)
    else
        drop_packet(pkt)
    end
else if fe.state == BPF then
    gk_bpf_decide_pkt(fe, pkt)
else
    drop_packet(pkt)
end
```

from the front to the back interface are expected to bypass ranges of IP addresses that should not go through Gatekeeper. For example, this can be used for incremental deployment; or enable a transit AS to bypass peering traffic destined yet to another AS. In those cases, Gatekeeper only needs to update the Ethernet header by changing its destination Media Access Control (MAC) address accordingly.

If the FIB entry instructs to take other actions, then the packet will be dropped (Lines 10 – 12). Finally, the function returns a flow entry *fe*, which is associated with the packet (Line 14). Note that this function will be only called for a request packet, as Gatekeeper will find a flow entry for packets associated with other states. After obtaining the flow entry, Algorithm 5 will process that packet based on the flow state by calling `process_flow_entry()`.

Algorithm 7 shows the pseudocode of the packet processing algorithm for packets associated with different flow states. More specifically, when a flow entry is at request state, all Gatekeeper does is to compute the priority of the packet, to encapsulate the packet as a request, and to put this encapsulated packet in the request queue managed by the Solicitor block (Lines $2 - 4$). Notice that the flow entry has the information about the last time that Gatekeeper sees a request packet from this flow. When a flow entry is at granted state, Gatekeeper processes the packet by verifying its capability stored in the flow entry (Line 6). If its capability is valid, then Gatekeeper encapsulates the packet as a granted packet (Line 7). Otherwise, the packet will be dropped (Line 9). When the flow entry is at BPF state, Gatekeeper calls the BPF program to determine the fate of the packet (*i.e.*, forward or decline) (Line 12). Once the flow is authorized by the BPF program, the policy decision has an option named `direct_if_possible`, when this option is set, packets belonging to this flow can be directly sent to the destination bypassing the Grantor servers. In this way, one would expect to reduce the load for the Grantor servers as the granted packets can be sent to the destinations bypassing the Grantor servers. When a flow entry is at other state, Gatekeeper processes the packet by initializing a flow entry and dropping the packet (Line 14).

4.5 Gatekeeper Optimization

4.5.1 Utilizing Modern NICs

Fast NICs, such as NICs with 10 Gbps and 100 Gbps speed, generate more packets than a single *lcore* can process in Gatekeeper. Notice that an *lcore* refers to a logical execution unit of the processor, sometimes called a *hardware thread*. Therefore, Gatekeeper needs to load balance these packets through a number of lcores to process all of them without any packet drops. To solve this issue, we investigated several techniques: (1) hardware-based solutions, like flow steering, implemented as Intel Ethernet Flow Director [Intel, 2015] and Receive-Side Scaling (RSS); and (2) software-based solutions, like DPDK packet distributor library, and the load balancer application [DPDK Project, 2018]. Because software-based solutions can introduce non-negligible overhead, especially with fast networks, and may even not guarantee in-order packet delivery, we choose the hardware-based solutions.

Our solution is to use multi-queue and RSS (see §4.5.1) to create a queue in the NIC for each lcore, and asks the NIC to hash the pair (source, destination) IP addresses of packets and place the packets in queues based on the hash values. Moreover, it uses hardware offloading (*e.g.*, IP and UDP checksum) to save CPU cycles, hardware filters to direct specific flows to specific queues for specific functional blocks, and hardware transactional memory for efficient synchronization if these hardware features are available. Otherwise it falls back to the software implementation. Notice that the hardware features are essentials for achieving high speed packet processing with minimum number of CPU cores. In the following sections, we will illustrate the key NIC hardware features.

NICs with RSS

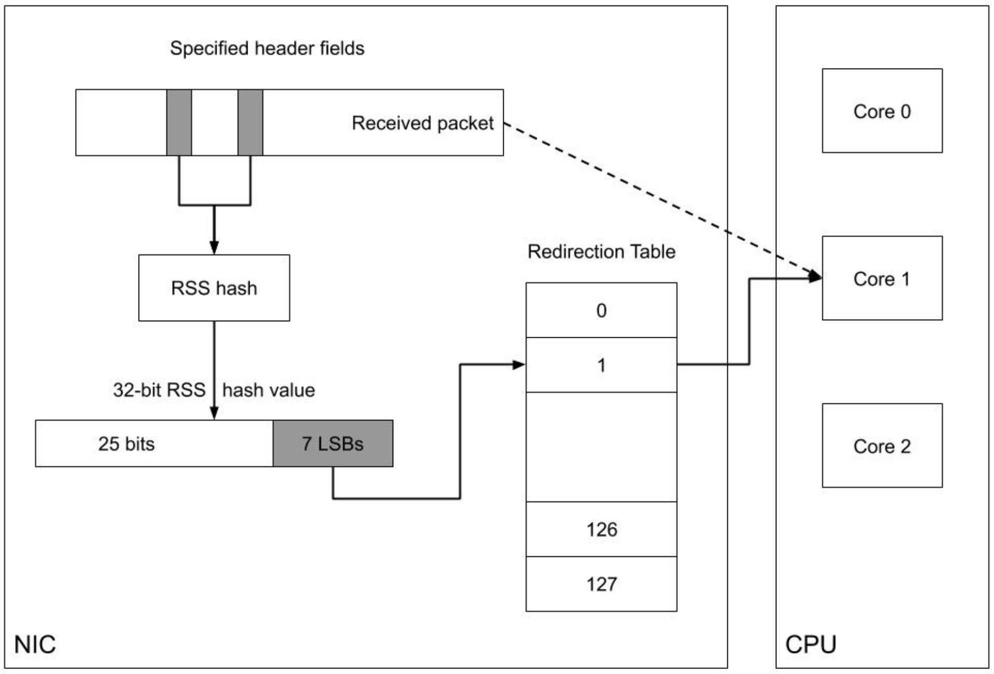


Figure 4·4: NIC RSS for received packets distribution.

RSS can efficiently distribute the network received packets across multiple CPU cores in a multi-core system. To fully utilize the potentials of the modern multi-core CPUs, we choose NICs with the multi-queue and RSS support as shown in Figure 4·4. The multi-queue support allows different CPU cores to access the NIC without contending with each other. In our example, we assume the *i*-th CPU core only processes the packets from the NIC queue *i*, indicated by the redirection table.

Specifically, a NIC with RSS support uses a RSS hash function to compute a hash value over a defined area, specified by a hash type, within the received network packet. The defined area can be non-contiguous. In Gatekeeper, the defined area is the IP pair of source and destination addresses. Then, seven least significant bits (LSBs) of the hash value are used to index a redirection table, which is configured when Gatekeeper boots. The values in the redirection table are used to assign the received packet to a queue, and the packet is further processed by the corresponding CPU core. In order to calculate the RSS hash value, the hash function needs to take a secret key of the RSS hash (RSK) as input. Gatekeeper randomizes the RSK in order to avoid hackers to know it. Interested readers can find the RSK generation algorithm (*i.e.*, the function `randomize_rss_key()`) from our Gatekeeper repository. From the example, we know that modern NICs with RSS support have two main advantages: (1) RSS enables the NIC to distribute a subset of incoming packets to different CPU cores; (2) RSS ensures that packets belonging to the same flow will be processed by the same lcore in the order that they arrive at Gatekeeper server, thus it avoids out-of-order packet delivery. Notice that the 32-bit RSS hash value calculated by the NIC hardware will be carried in the packet's metadata. As the flow hash table lookup will use this hash value for faster lookup, we extended the flow entry data structure to include this RSS hash value. It is essential to achieve high performance in hash table lookup and save significant amount of CPU cycles.

Other NIC features

It would be nice for the best performance if the NICs could support the following features in hardware: (1) L2 Ethertype filters: these filters identify packets by their L2 Ethertype and assign them to specific receive queues; (2) N-tuple filter: these filters identify specific L3/L4 flows or sets of L3/L4 flows; (3) Hardware offloading for IP and UDP headers checksum: as all packets that Gatekeeper decides to forward are encapsulated, and encapsulating packets should be done with minimum effort. These offloading features can save non-negligible amount of CPU cycles. (4) VLAN stripping for saving CPU cycles. We have already implemented the software version of these features, if Gatekeeper detects that they are not supported in hardware, it will fall back to their software equivalents. Another important feature expected from the mainboard/CPUs in order to reach the best performance is the hardware memory transactions (HTM), which increases the simplicity of synchronization without sacrificing performance.

4.5.2 Dealing with Hash Table Insertion Failures

Gatekeeper will always insert new flow entries into the flow hash table in the worst case, and insertion failures become common even as Gatekeeper aggressively scans over its flow hash table to remove expired flow entries. As Cuckoo hashing is commonly used by software systems (*e.g.*, CuckooSwitch [Zhou et al., 2013]) for high performance and has been implemented as part of the DPDK hash table library, we use it to manage the flow entries for Gatekeeper's first deployment. However, Cuckoo hashing is known to have great performance degradation when the hash table is near full [Herlihy et al., 2008]. This is because each flow entry has two candidate buckets, cuckoo hashing tries to repeatedly move flow entries around if the candidate buckets are full until one empty slot is found or the maximum number of moves (1000 as default in DPDK) has reached [Pagh and Rodler, 2001]. To solve the performance degradation issue, the key insight is that *having a single failure per batch may be bearable*. Given a batch of received packets, Gatekeeper tries to add all new flows, but if one addition fails for the batch of received packets, it terminates early and does not try all the other flows to be added in that batch. The reason is that under DDoS attacks, the resources of the server are limited, and Gatekeeper cannot insert all the flows into the hash table. To insert them successfully, Gatekeeper has to heuristically remove the existing flow entries from the table and tries to add the new flow entry again, which is quite costly. Instead, Gatekeeper gives this flow a chance sending a request to Grantor servers.

4.5.3 Group Prefetching (GP)

In Algorithm 5, Gatekeeper processes one packet each time. However, as modern CPUs allow multiple memory loads in flight at a time, this single packet processing suffers from low memory bandwidth utilization. As a consequence, the performance of Gatekeeper is severely restricted by the memory access latency. As group prefetching needs to split the code into different stages, a naive solution is to put all the code into one single function. However, in a complex system like Gatekeeper, each high-level component or code stage can be an independent module or even third-party libraries (*e.g.*, hash table library and LPM library) that implement their own data structures. The naive solution is unrealistic, thus we choose to apply group prefetching to each component or code stage separately.

As can be seen in Algorithm 5, the basic packet processing algorithm can be splitted into four code stages: (1) extract the packet information, including its IP flow information; (2) lookup the flow hash table; (3) lookup the LPM policy table; (4) process the packet as the flow entry instructs. An incoming packet does not necessarily have to go through all the four code stages. To improve the performance of Gatekeeper, we batch the packet processing with group prefetching in Algorithm 8. Note that we use a simplified version for the sake of presentation without considering many corner cases. For example, prefetching a `null` pointer could hurt performance. We use the DPDK memory prefetching function call `rte_prefetch0()` to prefetch a cache line into all cache levels. Moreover, we hide the implementation details of the batched lookup functions (*e.g.*, `batched_hash_table_lookup()`) and use a variable n as the batch size. One needs extra work to tune the best settings for parameters such as n. For the LPM policy table lookup, one can optimize it for only packets missing the hash table lookup.

Algorithm 8: Group prefetching-based packet processing algorithm

```
Input: A batch of packets received from the NIC: $pkts$
Input: Batch size: $n$
Input: A flow hash table: $flow\_hash\_table$
Input: An LPM table: $lpm\_tbl$
init flow_arr[n]  /* IP flow array placeholder */
init fe_arr[n]    /* flow entry array placeholder */
init fib_arr[n]   /* FIB entry array placeholder */
PREFETCH_OFFSET = 4
for $i \leftarrow 0$ to $PREFETCH\_OFFSET - 1$ do
    prefetch (pkts[i])
end
for $i \leftarrow 0$ to $n - PREFETCH\_OFFSET - 1$ do
    prefetch (pkts[i + PREFETCH_OFFSET])
    flow_arr[i] = extract_packet_info(pkts[i])
end
for $i \leftarrow n - PREFETCH\_OFFSET$ to $n - 1$ do
    flow_arr[i] = extract_packet_info(pkts[i])
end
fe_arr = batched_hash_table_lookup(flow_hash_table, flow_arr, n)
for $i \leftarrow 0$ to $n - 1$ do
    prefetch_flow_entry (fe_arr[i])
end
fib_arr = batched_lpm_fib_lookup(lpm_tbl, fe_arr, n)
for $i \leftarrow 0$ to $n - 1$ do
    prefetch (fib_arr[i])
end
for $i \leftarrow 0$ to $n - 1$ do
    if fe_arr[i] == null then
        fe_arr[i] = lookup_fe_from_lpm(flow_arr[i], pkts[i], fib_arr[i])
    end
    if fe_arr[i] != null then
        process_flow_entry(fe_arr[i], pkts[i])
    end
end
```

Algorithm 8 adds memory prefetching for the packet data, flow entry and FIB entry associated with the packet, before accessing it. The purpose of prefetching the packet data is to load the packet headers (up to the IP header) into the CPU cache, so that it can parse the packet faster (Lines 6 and 9). Basically, it issues a memory prefetch for each packet (64 bytes or 128 bytes depending on the hardware), which is enough to load an Ethernet header (14 bytes), an optional Ethernet VLAN header (8 bytes), and either an IPv4 header without options (20 bytes), or an IPv6 header without options (40 bytes). For IPv4 packet, the total size of the headers is as $14+8+20=42$ bytes), while for IPv6 packets, the total size is $14+8+40=62$ bytes. Notice that instead of issuing prefetches over all the packets in the batch before accessing them, the algorithm has a parameter, `PREFETCH_OFFSET` (Line 4), to control the software prefetching distance, so that one can expect to have timely prefetches. Otherwise, the demand request may arrive either early or late, leading to less effective prefetches. Moreover, issuing all the prefetches at the same time may become less effective when the number of issued prefetches in parallel exceeds the maximum number of prefetching streams that the CPU can handle at the same time. Therefore, algorithm 8 tries to interleave the memory prefetches for packet headers and the packet information extraction. To prefetch the flow entries, it needs to issue one or two prefetches depending on the hardware, as its size is greater than 64 bytes (Line 17). Notice that the only occasion that `lookup_fe_from_lpm()` returns a flow entry is when the corresponding FIB entry is at `FWD_GRANTOR` state. In this state, `initialize_flow_entry()` writes the flow entry without reading anything from the content of the flow entry. Therefore, the algorithm does not add any memory prefetching after calling `lookup_fe_from_lpm()`. Moreover, it issues a memory prefetching for the FIB entry (Line 21) when the packet misses the hash table lookup. Besides the above mentioned optimizations, Gatekeeper also adds memory prefetching for flow entries when periodically scanning over the flow hash table to remove the expired ones.

The main disadvantage of group prefetching is that when the processing of a packet finishes earlier than the final code stage, group prefetching just switches the processing from the current packet to the next packet without initializing the processing for a new packet. As a result, the number of code stages that group prefetching executes for all packets is equal to the maximum number of code stages that a packet needs to go through, regardless of whether the other packets have finished. Moreover, the best prefetching distance for each code stage can be quite different, how to tune the parameters to achieve the best performance as a system still needs further exploration. Finally, group prefetching suffers from control divergence issues [Kalia et al., 2015], which is tricky to split the code into different stages in a complex system.

4.5.4 Coroutines (Coro)

Coroutines are a valuable paradigm that enables programmers to do efficient context switching in software, and thereby achieve these quantitative and qualitative improvements: (1) improve applications' performance through memory access latency hiding; and (2) maintain high developer productivity, *i.e.*, the programmers no longer need to change the code structure, and they can add coroutines to existing code quickly and easily. Coroutines tend to bring extra overhead to applications as the applications need to keep a handler for each coroutine. Coroutines are a good match for Gatekeeper because, as we have seen, memory access is a frequent operation in Gatekeeper, and we anticipate the improvement from memory access latency hiding to be significant, and we do not expect to encounter a performance degradation in Gatekeeper due to the extra overhead.

To overcome the limitations of group prefetching, this section explores a coroutines-based framework to help programmer hide memory access latency efficiently. Although some programming languages (*e.g.*, C#, Python, Scala, and Javascript) have recently adopted coroutine-like `await` constructs, and C++ has merged the technical specification for coroutines as a language feature [ISO/IEC, 2017], the C programming language still does not have direct support for coroutines within the language or their standard libraries. As C language is the primary language used to implement Gatekeeper, we look for available libraries for coroutines written in C.

Based on the control transfer mechanism, coroutines can be classified into two types: symmetric and asymmetric coroutines. Symmetric coroutine facilities provide a single control-transfer operation that allows coroutines to explicitly pass control among themselves and they operate at the same hierarchical level. Asymmetric coroutine mechanisms provide two control-transfer operations: one for invoking a coroutine and one for suspending it, the latter returning control to the coroutine invoker. An asymmetric coroutine can be regarded as subordinate to its caller [Moura and Ierusalimschy, 2009]. As the symmetric coroutines minimize the switching overhead by transferring control directly to each other, we choose a library (*i.e.*, libcoro [Lehmann, 2018]) for symmetric coroutines.

Algorithm 9 presents the pseudocode of the coroutines-based packet processing algorithm in Gatekeeper servers. For the sake of presentation, it does not show the management logic of the coroutines. Basically, this algorithm is similar to the basic packet processing algorithm as shown in Algorithm 5, which processes a single packet. The difference is that whenever Gatekeeper needs to access a specific memory address, it issues a memory prefetch and then yields to the next coroutine. Each task for processing a packet is assigned to a coroutine. Whenever the coroutine finishes the current task, it would accept a new task for processing a new packet. In most cases, we can use the prefetch and yield function `f(addr, coro)` (*i.e.*, the last input parameter of Algorithm 9) to switch from the current coroutine to the next coroutine. This function simply takes the memory address (*i.e.*, `addr`) as well as the current coroutine (*i.e.*, `coro`) as input parameters, issues a memory prefetch by calling the DPDK function `rte_prefetch_non_temporal(addr)`, as there is little locality among the packets over the batch, and finally yields to the next coroutine which processes the next packet by calling `yield_next(coro)`. Notice that the `coro` structure has the necessary information to schedule the next coroutine. Moreover, we use a separate function `prefetch_flow_entry()` for prefetching the flow entry associated with the packet (Lines 8 and 9). For the two main lookup functions (Lines 6 and 12), we kept the original code structure and inserted prefetch and yield function only whenever there is an expensive memory access.

Algorithm 9: Coroutines-based packet processing algorithm

```
   Input: A packet received from the NIC: pkt
   Input: A flow hash table: flow_hash_table
   Input: An LPM table: lpm_tbl
   Input: A coroutine: coro
   Input: A prefetch and yield function pointer: f
 1 init flow  /* IP flow placeholder for the packet */
 2 init fe    /* flow entry placeholder for the packet */
 3 init fib   /* FIB entry placeholder for the packet */
 4 f(pkt, coro)
 5 flow = extract_packet_info(pkt)
 6 fe = hash_table_lookup_and_yield(flow_hash_table, flow, f, coro)
 7 if fe != null then
 8     prefetch_flow_entry (fe)
 9     yield_next(coro)
10 end
11 else
12     fib = lpm_fib_lookup_and_yield(lpm_tbl, fe, f, coro)
13     fe = lookup_fe_from_lpm(flow, pkt, fib)
14 end
15 if fe != null then
16     process_flow_entry(fe, pkt)
17 end
```

The best number of coroutines needed by Gatekeeper can vary depending on the workload. We proposed an algorithm to dynamically change the number of coroutines. Basically, the algorithm collects the average number of cycles for processing the tasks over the last batch of packets. When the workload changes, it changes the number of coroutines accordingly. Specifically, when it detects the last change does not bring any improvement, it goes back to the previous configuration, and changes the adjustment direction. Otherwise, it adjusts the number of coroutines based on the current adjustment direction. Note that each time the algorithm changes the number of coroutines by 1. Interested readers can find our implementation from our Gatekeeper GitHub repository [Machado et al., 2020].

4.5.5 Optimizing SOL Enqueue

In the initial implementation of the priority queue in SOL, the core data structure carrying a request packet is shown below:

```
struct priority_req {
    /* Doubly-linked list node. */
    struct list_head list;
    /* The packet for this request. */
    struct rte_mbuf *pkt;
    /* The priority of this request. */
    uint8_t priority;
};
```

Basically, each SOL instance uses a doubly-linked list to manage its priority queue. The field `list` in the above data structure allows SOL to manage it in the priority queue. The `pkt` field points to the actual request packet, carried by an `mbuf` instance. Note that the `mbuf` library in DPDK provides the ability to allocate and free buffers (mbufs) that may be used by the DPDK application to store message buffers. A new request will be appended to the end of the appropriate priority specified by `priority` field. Whenever Gatekeeper tries to enqueue requests to the priority queue of SOL, it needs to allocate `priority_req` entries. Instead of allocating the entries on-the-fly, we keep a static memory pool of the `priority_req` entries in SOL by using DPDK's `mempool` library. Whenever Gatekeeper needs to enqueue entries, it first tries to obtain an entry from SOL's memory pool, and fill up the fields accordingly. Under the worst case, all the incoming packets are requests, the allocation operations have very high memory loads according to Linux perf profiling.

This optimization tries to remove the use of `struct priority_req`, and make the packet itself carry all the necessary information so that Gatekeeper can simply use the lockless ring to transfer the requests to SOL directly. Fortunately, DPDK allows programmers to embed private data (user data) in a given mbuf when creating a packet mbuf pool. Thus, it uses the private data of the mbuf to carry the list information. Moreover, in mbuf, it has a field `udata64`, which allows a mbuf to carry user data, so we use this field to carry the priority information. With this optimization, it removes the need for `struct priority_req` as well as the `mempool` holding the entries.

To further improve Gatekeeper's performance, instead of enqueuing one request packet each function call, Gatekeeper enqueues a batch of request packets once after processing all the packets in the batch.

4.5.6 Hash Tables

The flow hash table is on the critical path, because every packet needs to lookup the hash table, and may need to insert a new flow entry into the table if the packet is a new request. The current flow tables use the DPDK hash library implementation of Cuckoo hashing. While this library provides Gatekeeper with a good starting point, it has some edge cases that conflict with the way that Gatekeeper uses it. The two salient points are the following: (1) DPDK hash tables have a key store that maintains a copy of all the keys present in the tables. These keys are needed to ensure that a lookup does not confuse two entries with identical hash values. The key store also eliminates the need for entries having a copy of their keys. Nevertheless, when entries must have a copy of their keys, like flow entries in Gatekeeper, the key store brings two disadvantages: (a) duplicate, wasted memory; and (b) higher cache misses when doing entry lookup. (2) while DPDK hash tables have a fast lookup, they struggle to add new entries when the table has high occupancy. Specifically, as a flow key in Gatekeeper is 36 bytes, and assuming a flow table has 2^{20} entries, that is, the key store takes 36MB of duplicate memory, which is larger than the typical CPU LLC size. Moreover, one of the lasts steps of an entry lookup is to load the key being searched. When the number of entries with the same hash value is small and the flow table is large, it is better to load the entry instead of the key to minimize cache misses in the running core. In Gatekeeper, we conducted extensive experiments, and the results show that less than 1% of the buckets have entries with the same hash value. Finally, the insertion performance of the flow hash table is critical for Gatekeeper during an attack that explores the keyspace.

A feasible solution to this problem must balance the faster lookup speed with the occupancy of the table. To explore the trade-off between lookup speed and table occupancy, we investigated six promising hash table algorithms: linear probing [Knuth, 1997], hopscotch hashing [Herlihy et al., 2008], separate chaining (implemented as a linked list), double hashing [Knuth, 1973], quadratic probing and cuckoo hashing [Pagh and Rodler, 2001]. The experimental results as shown in the following section can guide us to redesign the flow hash table in Gatekeeper to further improve its performance.

4.6 Experimental Results

In previous sections, we have discussed the design of Gatekeeper. Especially, we focused on the packet processing in Gatekeeper as well as the optimization techniques to improve its performance. This section provides experimental results on Gatekeeper and shows how each of the proposed optimizations contributes to Gatekeeper's performance.

4.6.1 Experimental Setup

Testbed Setup

We conduct experiments on a server equipped with two Intel Xeon E5-2690 V2 CPUs (10 cores, 25MB LLC, 3.00GHz). Figure 4·5 shows the topology of NIC ports and CPUs on

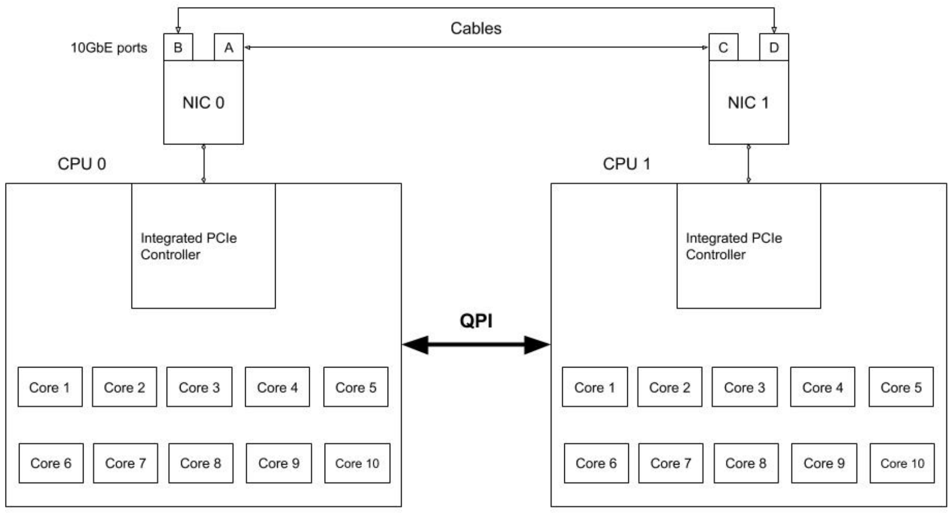


Figure 4·5: Topology of NIC ports and CPUs.

our server. The two CPUs are connected via an Intel QuickPath Interconnect (QPI). With Hyper-Threading enabled, we can effectively have 20 threads per CPU. The server has two dual-port Intel 82599 [Intel, 2019b] 10GbE NICs, and runs 64-bit Ubuntu 16.04 LTS. Assuming ports A and B belong to the same NIC, while ports C and D belong to another NIC. Ports A and C are connected with each other, while ports B and D are connected. Gatekeeper runs on port C as the front interface and port D as the back interface. We uses DPDK's NUMA-aware memory allocation with hugepages (see §2.3.2) enabled for all data structures in its implementation. Specifically, we use 2MB pages.

To evaluate the effectiveness of each optimization technique, we run Gatekeeper's packet processing on a single lcore. To saturate that lcore, we launch 16 generators running on port A, so that the generated traffic will be sent to port C and further processed by Gatekeeper. Each generator uses a Linux raw socket to generate minimum-sized 64 bytes UDP packets with the same destination address and random source address. Note that after

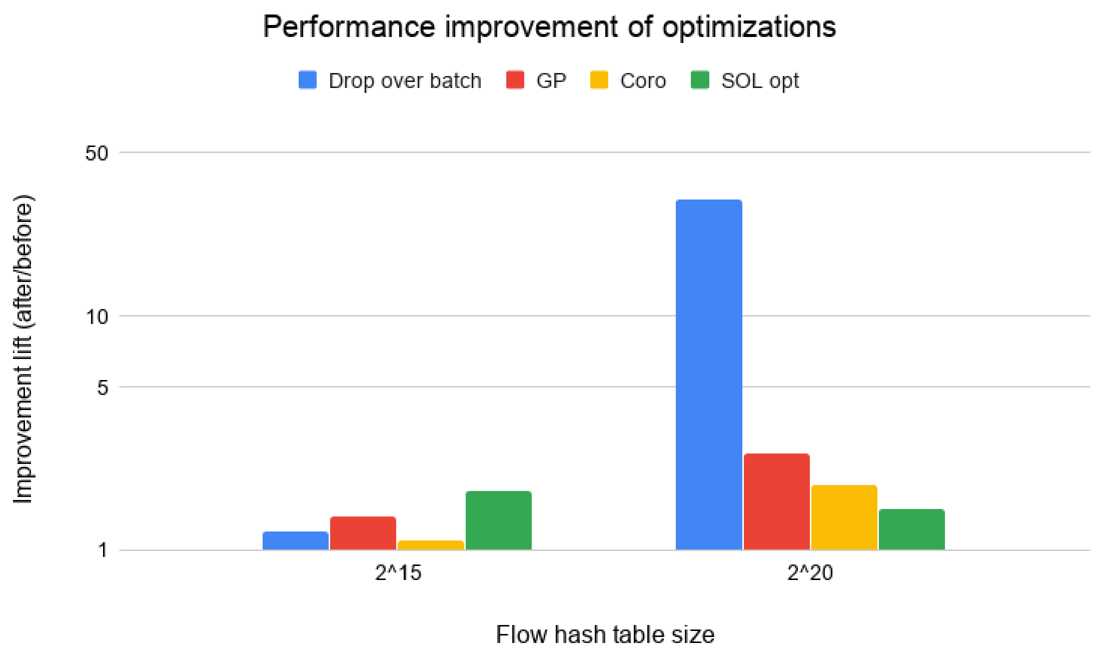


Figure 4·6: Performance improvement of optimizations.

Gatekeeper boots, the test script will add a policy to Gatekeeper's policy table, so that the traffic will have to request permission from Grantor. For each experiments, we collected the statistics every 30 seconds, and get the average results over a period of 300 seconds (*i.e.*, average over 10 data points). For the detailed description of our experimental setup, please refer to our wiki under the "Experimenting with Gatekeeper" section [Machado et al., 2020].

4.6.2 Performance Evaluation on Testbed

Gatekeeper Performance

Figure 4·6 shows the performance improvement of the key optimizations as described in the previous sections. The Y-axis is logscaled. In the experiment, we consider two flow hash table sizes: 2^{15} and 2^{20}. As we can see, dropping new requests over the batch improves Gatekeeper's performance by $1.2\times$ for the small hash table and $31.7\times$ for the large hash table. Note that we see better result for the large hash table, this is due to two main reasons: (1) the average number of buckets that Gatekeeper needs to iterate through when inserting a new flow entry is larger for the large hash table; (2) the large hash table causes more cache misses when doing table lookup.

The second optimization is group prefetching. Group prefetching improves Gatekeeper's performance by 1.4× and 2.6× for the two settings, respectively. Again, we see better result for the large hash table because group prefetching can reduce more cache misses for large hash table by hiding more memory access latency. Similarly, the coroutines-based optimization improves Gatekeeper by 1.1× and 1.9× for the small and large hash tables, respectively. Note that the coroutines-based optimization performs a little bit worse than group prefetching, as in our experiments, almost all the packets will go through all the stages to the final code stage, leaving little to no room for improvement. However, the coroutines-based framework needs to maintain more state than group prefetching.

The fourth optimization is to optimize the SOL enqueue by batching and designing compact data structures. It improves Gatekeeper's performance by 1.8× and 1.5× for the two settings, respectively.

Note that when having larger tables (*e.g.*, table with 2^{25} entries), the results are similar. By applying all the optimization techniques, Gatekeeper's performance was improved by more than 90× compared to the vanilla baseline implementation.

Hash Table Performance

We evaluate the lookup speed of the above mentioned six hashing algorithms on a single core using synthetic data set. The *load factor* of a hash table is the number of keys stored in the hash table divided by the size of the hash table. We vary the load factor of the hash tables from 0.1 to 0.9. Note that we run each set of experiments for 1000 times and get the average lookup speed. Through the performance evaluation on various hashing algorithms

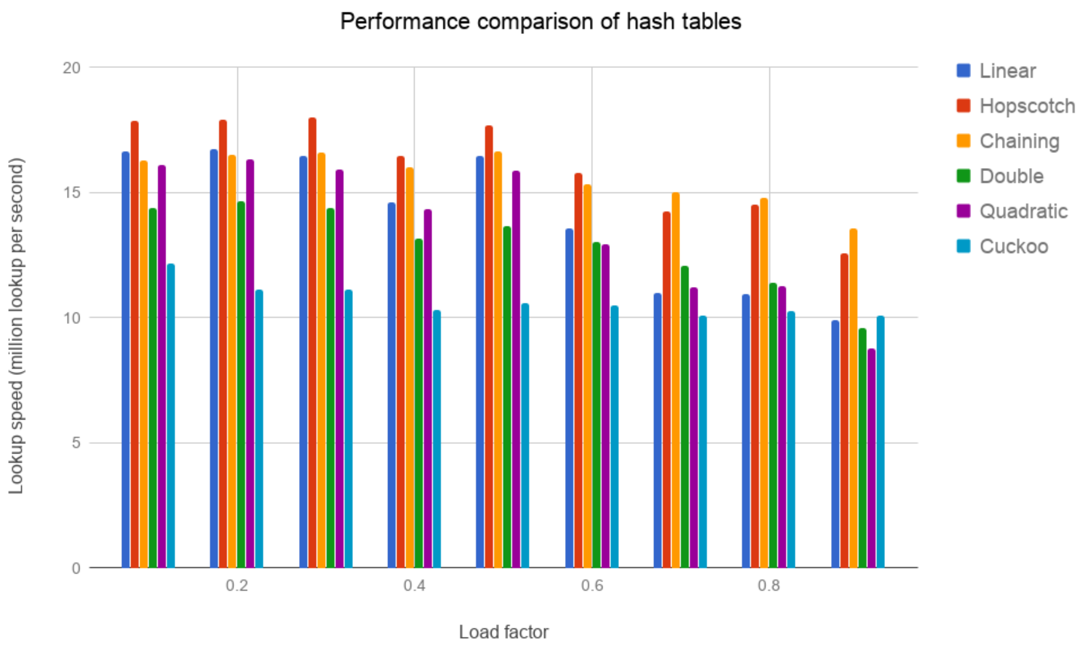


Figure 4·7: Hash table lookup performance.

as shown in Figure 4·7, hopscotch hashing and chaining are good candidates.

Moreover, when considering the insertion speed as well, the separate chaining will be better as it simply inserts the new item into the head of the linked list identified by the hash value of the key. We would like to redesign the flow hash table in Gatekeeper and further explore the trade-offs in the Gatekeeper system under a realistic workload.

4.7 Summary

In this chapter, we place our focus on improving the performance of Gatekeeper, the first open-source and deployable DoS mitigation system. Gatekeeper is implemented based on Intel's DPDK, a set of libraries for high-speed packet I/O. We present a series of optimization techniques, including group prefetching and coroutines, to improve Gatekeeper's performance. Compared with vanilla Gatekeeper implementation, we improve its performance by a factor of more than 90×. These optimizations make Gatekeeper ready for the first deployment with a requirement of 10 Gbps throughput. Since the ultimate goal of Gatekeeper is to achieve 100+ Gbps throughput on a single commodity server, and the flow hash table is on the critical path, this thesis further conducted experiments on various hashing algorithms to evaluate their performance. The results show that hopscotch hashing and separate chaining outperform the others and are promising candidates for a new flow hash table in Gatekeeper.

Chapter 5

Conclusions and Future Work

The previous chapters have introduced several acceleration techniques by taking advantage of modern CPU features to improve the performance of software packet processing applications. Then, we dive into two concrete applications: IP lookup and DDoS mitigation. Specifically, we proposed SAIL, a splitting approach to IP lookup, and a suite of algorithms for IP lookup based on SAIL framework. Experimental results show that our SAIL algorithms are much faster than the state-of-the-art IP lookup algorithms. We also presented Gatekeeper, the first open source and deployable DDoS mitigation solution. With a series of optimization techniques, we significantly improved packet processing performance of Gatekeeper over the vanilla baseline implementation.

The remainder of this chapter concludes with our final remarks in §5.1, and outlines future work on both IP lookup and Gatekeeper in §5.2.

5.1 Conclusions

Driven by cloud computing, we have seen and continue to see the replacement of dedicated hardware appliances with software-based solutions. With the end of Moore's law, the performance gain from a single CPU core will diminish over time. However, the network link speed still grows rapidly. Therefore, we cannot solely rely on the performance improvement of a single CPU core to deal with the ever-growing network link speed. Instead, we have to find a viable path to achieve high-performance software packet processing. Over the past few years, we have made consistent efforts to improve the performance of software packet processing in network applications. Through extensive practice, we gained a lot of experience in this area. To achieve the highest potential performance for software packet processing, we have to optimize the network applications in a holistic way. Below we summarize five principles that generally guided the performance improvement of software packet processing during our work.

First, *design network applications by taking advantage of parallelism*. Modern CPUs typically have multiple cores, and modern NICs with advanced features (*e.g.*, multi-queue and RSS) allow developers to take advantage of the multi-core CPUs. Moreover, modern NUMA platforms typically support multiple sockets (*e.g.*, 2, 8, 32), and these platforms can be easily scaled up to deal with the increasing workloads. Finally, to avoid the contention among different CPU cores and better utilize the cache/memory locality, we should carefully design the granularity (*e.g.*, per-queue, per-NIC, per-CPU, per-socket, or global) of the data structures.

Second, *design new data structures by taking modern CPU features and applications' workload into account*. Modern CPUs have multiple levels of caches, and their cache sizes are increasing. New data structures for fast packet processing should be compactly designed to increase the CPU cache utilization as well as the effective memory bandwidth. The principle behind our two dimensional splitting in our SAIL framework can be generalized to the design of new data structures. Specifically, when designing a new data structure, we should always ask the key question: *whether can we split the data structure into different parts, so that it can optimally utilize the CPU cache hierarchy by placing some parts into the caches?* Modern CPUs also support advanced instructions, such as SIMD, to improve the throughput of applications. For example, Poptrie uses the *POPCNT* instruction in CPU to achieve high performance.

Moreover, from our experiments in Gatekeeper, we see that hash table plays an important role in Gatekeeper. It is also true for many other applications, such as network measurement and in-memory key value store. The workload can affect the performance of hash tables significantly. Although the DPDK hash table library using cuckoo hashing has broad applications, it does not work well under write-intensive workload. An ideal hash table algorithm should meet with the following three challenging requirements at the same time: high lookup speed, high insertion speed, and high table occupancy. We have done some preliminary experiments to compare various hash table algorithms and would like to see some future work on this topic.

Third, *hide memory access latency in software.* As the performance gap between CPU and memory still remains large, there is a strong demand for optimization techniques to hide memory access latency. Although many efforts tried to hide memory latency (see §2.4.2) in hardware, they have limited effectiveness due to the hardware limit. An ideal solution for software packet processing should achieve high performance while maintaining very high developer productivity. We introduced the coroutines-based solution to the network community, however, this solution is still in its early stages. In the future, we would like to see more efforts from different communities, such as the network community and compiler community, to make it a paradigm for high-performance software packet processing.

Fourth, *think about NICs even when the processing is software-based.* With the advancement of modern NICs, they support more and more on-board hardware resources to offload packet processing tasks. For example, modern NICs typically have specialized hardware for computation-intensive network functions, such as encryption/decryption, encapsulation/decapsulation, IP/UDP header checksum. As the hardware is fast enough to execute these network functions for all packets. We should co-design the application and NIC hardware, and offload these tasks to the NICs whenever the NICs support these offloading. Thus, the CPU cores can have more room to do other important tasks. We will demonstrate our design choices on IP lookup as well as Gatekeeper by taking advantage of SmartNICs in §5.2.

Fifth, *improve performance carefully guided by system profiling*. Modern network applications typically have many components that run on a NUMA platform. Every part of the system (*e.g.*, PCIe, memory bandwidth, cache misses, CPU cycles, context switches, synchronization, communication, etc.) can potentially become the bottleneck. We have to carefully improve the packet processing performance guided by system measurement using profiling tools, such as Intel VTune Profiler [Intel, 2019c] and Linux perf [Linux, 2009]. We extensively used Linux perf to profile Gatekeeper system. Almost every optimization technique that we came up was through careful system profiling. For example, we found that the hash computation in flow hash table lookup initially dominated the CPU cycles, then we realized that by taking advantage of the RSS hash value of the flows calculated by the NICs can significantly reduce the CPU cycles. Many other optimizations that are not covered in thesis, including NUMA-aware allocation, the appropriate locations to insert memory prefetch instructions as well as the prefetching distance in group prefetching.

5.2 Future work

For SAIL, it currently can deal with multiple hundreds of gigabits per second network traffic on a single CPU core using the lookup optimized version SAIL_L. Although its average update complexity is low, the worst case is unbounded. Ideally, we need to optimize SAIL_L, so that it can have constant yet fast update speed as well. Moreover, it is desirable to implement SAIL on SmartNICs, so that IP lookup can be done without involving the host CPUs. For Gatekeeper, since its ultimate goal is to achieve 100+ Gbps throughput on a single commodity server, we may have to offload some of its tasks to specialized hardware. We discuss the future directions on how Gatekeeper can utilize ASIC-base SmartNICs to achieve its goals. Also, we discuss several other optimization techniques to further improve its packet processing performance.

5.2.1 IP Lookup Optimizations

Updating Optimization for SAIL_L

A trie node with a next hop can be represented by, and replaced by its two child nodes with the same next hop. Based on this regulation, leaf pushing [Srinivasan and Varghese, 1999] pushes the internal nodes with next hops to leaf nodes, and there is no internal node with next hops in the trie after leaf-pushing, thus the prefix overlap is eliminated. For example, node A with next hop n_p can be pushed to A's two child nodes with next hop n_p. When A's next hop is updated, the two child nodes also need to be updated. Currently, SAIL_L needs 2 off-chip memory accesses for lookup in the worst case, however, its update complexity is unbounded in the worst case. Although we have proposed an optimization alogrithm (*i.e.*, SAIL_U) to make a tradeoff between lookup and update, the lookup process needs 4 off-chip memory accesses in the worst case, twice as SAIL_L. A key question is: *can we achieve small constant on-chip memory usage, constant yet fast lookup speed, as well as constant yet fast update speed at the same time?* To meet these challenging requirements, it is critical to avoid the domino effect in leaf pushing. We are working on a new algorithm named `virtual pushing` to perform both fast lookup and fast update.

Offloading SAIL to SmartNICs

The on-board CPUs of SmartNICs become more and more powerful, and some of them have cache sizes that are far more than 4 MB, which is the maximum cache size needed by SAIL. Moreover, SmartNICs have enough DRAM to hold the necessary data structures for SAIL (*e.g.*, next hop arrays). As some SmartNICs support TCAM-style match-action policies, SAIL can potentially use them to deal with prefixes with length longer than 24. It would be interesting to explore the potential implementation choices for SAIL on these SmartNICs and evaluate their performance. It is highly likely that SAIL could keep up with the ever-growing link speed with only a small number of on-board CPU cores. If so, the IP lookup can be offloaded to these SmartNICs for better performance with lower cost.

5.2.2 Gatekeeper Optimization: Roadmap to 100+ Gbps Deployment

Gatekeeper Acceleration using SmartNICs

Inspired by the recent work on SmartNICs [Siracusano and Bifulco, 2017, Li et al., 2017, Le et al., 2017, Liu et al., 2019a, Moon et al., 2020], we can further improve the performance of Gatekeeper to achieve 100+ Gbps throughput on a single commodity server by taking advantage of SmartNICs. Recently, major network hardware vendors have released different SmartNIC products, in which a single port can deliver up to 100 Gbps throughput, such as Mellanox BlueField [Mellanox, 2020], Netronome Agilio LX [Netronome, 2018]. However, it is generally challenging to have a fully offloaded solution due to the limited memory and computation capabilities of the current SmartNICs compared to current high-performance servers.

To better co-design Gatekeeper and the SmartNICs, we need to investigate the key question: *which parts of Gatekeeper can be offloaded to the SmartNICs and which parts should be processed by the host CPUs*. For example, some SmartNICs have specific network accelerators for fixed network functions (*e.g.*, checksum, encryption/decryption, deep packet inspection), we can offload these compute-intensive functions of Gatekeeper to the SmartNIC. Moreover, as the flow hash table management is on the critical path of Gatekeeper, we can selectively offload part of its application logic to the SmartNIC. One feasible solution is to offload the flows that either will be declined by Gatekeeper or will bypass Gatekeeper to the ASIC-based data path. For example, the Netronome Agilio LX 100GbE SmartNIC [Netronome, 2018] supports flexible exact/wild card match-action policy offloads with up to 8 million flows on card. As some high-end SmartNICs have their own on-board DRAM, and some may support eBPF [Fleming, 2017, Cilium Authors, 2018] offload, we can come up *a heuristic flow offloading algorithm* to choose a number of flows in other states to offload their processing to the SmartNIC. A Bloom filter can be designed to decide whether the flow should be processed by the SmartNIC or not. The Bloom filter can easily support millions of flows while residing in the SmartNIC's CPU cache for high performance. Also, the hash values needed by the Bloom filter can be calculated using the SmartNIC's network accelerators.

Notice that many factors may affect the design of the heuristic flow offloading algorithm. For example, (1) the communication overhead between the SmartNIC and its host can affect the update speed of the flow tables in the SmartNIC [Miano et al., 2019]; (2) the CPU cache size, and the DRAM size, can limit the maximum number of flows that can be offloaded to the SmartNIC; (3) the SmartNIC CPU configuration (*e.g.*, the number of CPU cores, frequency), and the communication overhead between the SmartNIC CPU and its on-board DRAM, can limit the total percentage of the traffic that can be offloaded to the SmartNIC; (4) the traffic workload can change rapidly, for instance, we may see more request flows when under DDoS attacks. How to quickly adapt to the workload change is also critical. With all these constraints in mind, we should first choose a few potential SmartNICs that are suitable for Gatekeeper, then characterize the performance metrics (*e.g.*, SmartNIC on-board DRAM access latency) of the SmartNICs that can affect the design of the flow offloading algorithm, and finally, design the flow offloading algorithm on the chosen SmartNIC. With the rapid advancement of the SmartNIC on-board hardware resources (*e.g.*, Marvell OCTEON TX2 SoCs can scale up to 200 Gbps [Marvell, 2020]), we expect to improve Gatekeeper's performance significantly using SmartNICs, and we leave the further exploration on this direction as future work.

Implementing SAIL for IP Lookup in Policy Table

The policy tables in both Gatekeeper and Grantor servers are implemented using DPDK's LPM library, which uses a variation of the DIR-24-8 algorithm. Through the evaluation in the DXR paper [Zec et al., 2012], we know that the throughput of DIR-24-8 flattens as the number of parallel worker threads grows: the large amount of memory used causes frequent accesses to off-chip DRAM which saturates with parallel worker thread. To achieve 100+ Gbps throughput on a single server, we can use SAIL [Yang, 2014].

Generic Receive Offload (GRO)

GRO is an offloading technique that used to reassemble small packets into larger ones, therefore, it can potentially reduce the per-packet processing overheads. This optimization has the potential to save amount of CPU cycles, which is essential to achieve high speed network processing.

Batching Packets for Lua Policy Lookups in Grantor

In our current implementation, Grantors ask the Lua policy for policy decisions one packet at a time. Sophisticated policies can take advantage of the batched packets to perform better. For example, policies that use LPM tables to classify addresses could batch lookups of LPM tables. Another example is that when a number of packets over the batch have the same signature (*e.g.*, destination IP address) for policy decision lookup, Grantor only needs to do one lookup for those packets instead of doing one lookup per packet, thus it avoids the duplicate lookups.

Combining SAIL and SmartNICs to Accelerate Gatekeeper

Due to the hardware limitations of SmartNICs, we cannot offload the whole flow table to SmartNICs. Compared to SmartNICs with several gigabytes of memory, the host can support terabytes of memory. Thus, most parts of the flow table will reside in the main memory. For the LPM policy table, SmartNICs potentially have enough resources to offload it. I foresee once we have the SAIL algorithm in SmartNICs, we can completely offload the LPM policy table lookup in Gatekeeper to SmartNICs. The saved CPU cycles on the hosts can be used to process more packets (*i.e.*, higher throughput). With SAIL offloading, we can further split the work between SmartNICs and Gatekeeper hosts. Specifically, if an incoming packet needs to be either dropped or forwarded to the back network according to the LPM policy on the packet's destination IP address, then the packet will be processed by SmartNICs directly. Otherwise, SmartNIC transfers the packet along with the hash value as well as the LPM policy table lookup results to the host for further processing. Therefore, the hosts only need to process flows under Gatekeeper's protection. This optimization is well aligned with all the suggestions we had before in this chapter, and can be used to extend those suggestions.

After the LPM policy table lookup is removed from the packet processing on Gatekeeper hosts, we can streamline the packet processing on the hosts for further optimization. Specifically, for a request packet that has no existing flow entry associated with it in the flow table, Gatekeeper currently needs to conduct one flow table lookup, one LPM policy table lookup, and one flow table insertion. After removing the LPM table from the packet processing on Gatekeeper hosts, Gatekeeper can use the hash value to look up the flow hash table, and directly insert a new flow entry, initialized with the policy information (*e.g.*, IP address of the Grantor server), into the flow table. That is to say, we can get the lookup and insertion at the same time for a request, so Gatekeeper avoids the second lookup into the flow table. For a packet belonging to an established flow, Gatekeeper simply looks up the flow table to find its associated flow entry, and then enforces policy on the flow without the need for LPM policy table lookup. Therefore, the LPM table lookup on SmartNIC is wasted in this case. A requirement is that SAIL in SmartNICs must be able to process packets at line rate or higher. Otherwise, packet drops will happen. With the above optimization, we would expect that the worst-case performance of Gatekeeper can be doubled, although it brings extra overhead on SmartNICs for established flows.

References

Akhbarizadeh, M. J., Nourani, M., Panigrahy, R., and Sharma, S. (2006). A TCAM-based parallel architecture for high-speed packet forwarding. *IEEE Transactions on Computers*, 56(1):58–72.

Alipourfard, O., Moshref, M., Zhou, Y., Yang, T., and Yu, M. (2018). A comparison of performance and accuracy of measurement algorithms in software. In *Proceedings of the Symposium on SDN Research*, pages 1–14.

Almási, G., Caşcaval, C., Castaños, J. G., Denneau, M., Lieber, D., Moreira, J. E., and Warren Jr, H. S. (2003). Dissecting Cyclops: A detailed analysis of a multithreaded architecture. *ACM SIGARCH Computer Architecture News*, 31(1):26–38.

Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. (1990). The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6.

Andersen, D. G. et al. (2003). Mayday: Distributed Filtering for Internet Services. In *USENIX Symposium on Internet Technologies and Systems*, volume 4.

Anderson, T., Roscoe, T., and Wetherall, D. (2004). Preventing Internet Denial-of-Service with capabilities. *ACM SIGCOMM Computer Communication Review*, 34(1):39–44.

Anstee, D., Bowen, P., Chui, C., and Sockrider, G. (2017). 12th Annual Worldwide Infrastructure Security Report. `https://www.netscout.com/news/press-release/worldwide-infrastructure-security-report`.

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., et al. (2017). Understanding the Mirai Botnet. In *26th USENIX Security Symposium (Security 17)*, pages 1093–1110.

Argyraki, K. J. and Cheriton, D. R. (2005). Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks. In *2005 USENIX Annual Technical Conference (ATC 05)*, volume 38.

Asai, H. and Ohara, Y. (2015). Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup. *ACM SIGCOMM Computer Communication Review*, 45(4):57–70.

Baboescu, F., Tullsen, D. M., Rosu, G., and Singh, S. (2005). A tree based router search engine architecture with single port memories. In *32nd International Symposium on Computer Architecture (ISCA 05)*, pages 123–133.

Bando, M. and Chao, H. J. (2010). Flashtrie: Hash-based prefix-compressed trie for IP route lookup beyond 100Gbps. In *Proceedings of IEEE INFOCOM*, pages 1–9.

Barbette, T., Soldani, C., and Mathy, L. (2015). Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16.

Basescu, C., Reischuk, R. M., Szalachowski, P., Perrig, A., Zhang, Y., Hsiao, H.-C., Kubota, A., and Urakawa, J. (2015). SIBRA: Scalable Internet Bandwidth Reservation Architecture. *arXiv preprint arXiv:1510.02696*.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95.

Braun, P. and Litz, H. (2019). Understanding Memory Access Patterns for Prefetching. In *International Workshop on AI-assisted Design for Architecture (AIDArc)*.

Callahan, D., Kennedy, K., and Porterfield, A. (1991). Software prefetching. *ACM SIGARCH Computer Architecture News*, 19(2):40–52.

Cerović, D., Del Piccolo, V., Amamou, A., Haddadou, K., and Pujolle, G. (2018). Fast packet processing: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):3645–3676.

CERT Coordination Center (1996). TCP SYN Flooding and IP Spoofing Attacks. `https://resources.sei.cmu.edu/asset_files/WhitePaper/1996_019_001_496172.pdf`.

Chen, S., Ailamaki, A., Gibbons, P. B., and Mowry, T. C. (2007). Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3).

Chen, T.-F. and Baer, J.-L. (1994). A performance study of software and hardware data prefetching schemes. *ACM SIGARCH Computer Architecture News*, 22(2):223–232.

Choi, S., Shahbaz, M., Prabhakar, B., and Rosenblum, M. (2019). λ-NIC: Interactive Serverless Compute on Programmable SmartNICs. *arXiv preprint arXiv:1909.11958*.

Cilium Authors (2018). BPF and XDP Reference Guide. `https://cilium.readthedocs.io/en/latest/bpf/`.

Cisco (2020). Cisco Annual Internet Report (2018–2023). https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf.

Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408.

Crescenzi, P., Dardini, L., and Grossi, R. (1999). IP Address Lookup Made Fast and Simple. In *European Symposium on Algorithms*, pages 65–76. Springer.

Dai, H., Meng, L., and Liu, A. X. (2018). Finding persistent items in distributed datasets. In *Proceedings of IEEE INFOCOM*.

Dai, H., Shahzad, M., Liu, A. X., and Zhong, Y. (2016a). Finding persistent items in data streams. *Proceedings of the VLDB Endowment*, 10(4):289–300.

Dai, H., Zhong, Y., Liu, A. X., Wang, W., and Li, M. (2016b). Noisy Bloom Filters for Multi-Set Membership Testing. In *Proceedings of the ACM SIGMETRICS*, pages 139–151.

Dalton, M., Schultz, D., Adriaens, J., Arefin, A., Gupta, A., Fahs, B., Rubinstein, D., Zermeno, E. C., Rubow, E., Docauer, J. A., et al. (2018). Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387.

Datapath.io (2017). Cloud to Cloud Network. https://datapath.io/.

Dean, J. (2019). The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design. *arXiv preprint arXiv:1911.05289*.

Degermark, M., Brodnik, A., Carlsson, S., and Pink, S. (1997). Small forwarding tables for fast routing lookups. *ACM SIGCOMM Computer Communication Review*, 27(4):3–14.

Dharmapurikar, S., Krishnamurthy, P., and Taylor, D. E. (2003). Longest Prefix Matching Using Bloom Filter. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 201–212.

Dixon, C., Anderson, T. E., and Krishnamurthy, A. (2008). Phalanx: Withstanding Multimillion-Node Botnets. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, volume 8, pages 45–58.

Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. (2009). Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 15–28.

DPDK Project (2010). Intel Data Plane Development Kit. http://www.dpdk.org/.

DPDK Project (2018). DPDK Load Balancer Sample Application. `https://doc.dpdk.org/guides-18.08/sample_app_ug/load_balancer.html`.

DPDK Project (2019). DPDK Intel NIC Performance Report Release 19.08. `https://fast.dpdk.org/doc/perf/DPDK_19_08_Intel_NIC_performance_report.pdf`.

DPDK Project (2020). Berkeley Packet Filter Library. `https://doc.dpdk.org/guides/prog_guide/bpf_lib.html`.

Eatherton, W., Varghese, G., and Dittia, Z. (2004). Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122.

Fadishei, H., Zamani, M. S., and Sabaei, M. (2005). A novel reconfigurable hardware architecture for IP address lookup. In *2005 Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 81–90. IEEE.

Fan, B., Andersen, D. G., and Kaminsky, M. (2013). Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384.

Fatahalian, K. and Houston, M. (2008). A closer look at GPUs. *Communications of the ACM*, 51(10):50–57.

Fayaz, S. K., Tobioka, Y., Sekar, V., and Bailey, M. (2015). Bohatei: Flexible and elastic DDoS defense. In *24th USENIX Security Symposium (Security 15)*, pages 817–832.

Feo, J., Harper, D., Kahan, S., and Konecny, P. (2005). Eldorado. In *Proceedings of the 2nd Conference on Computing frontiers*, pages 28–34.

Ferguson, P. and Senie, D. (2000). IETF RFC2827: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. `https://tools.ietf.org/html/rfc2827`.

Firestone, D. (2017). VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328.

Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al. (2018). Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66.

Fleming, M. (2017). A thorough introduction to eBPF. `https://lwn.net/Articles/740157/`.

Floyd, S. and Jacobson, V. (1993). Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413.

Fu, J. and Rexford, J. (2008). Efficient IP-address lookup with a shared forwarding table for multiple virtual routers. In *Proceedings of the 4th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 08)*, pages 1–12.

Fu, J. W., Patel, J. H., and Janssens, B. L. (1992). Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter*, 23(1-2):102–110.

Fuller, V., Li, T., Yu, J., and Varadhan, K. (1993). Classless inter-domain routing (CIDR): an address assignment and aggregation strategy. `https://tools.ietf.org/html/rfc1519`.

Gai, S. (2020). *Building a Future-Proof Cloud Infrastructure: A Unified Architecture for Network, Security, and Storage Services*. Pearson Education.

Go, Y., Jamshed, M. A., Moon, Y., Hwang, C., and Park, K. (2017). APUNet: Revitalizing GPU as Packet Processing Accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 83–96.

Greenhalgh, A., Handley, M., and Huici, F. (2005). Using Routing and Tunneling to Combat DoS Attacks. In *USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*.

Guerzoni, R. et al. (2012). Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, volume 1, pages 5–7.

Gupta, P., Lin, S., and McKeown, N. (1998). Routing lookups in hardware at memory access speeds. In *Proceedings of IEEE INFOCOM*, volume 3, pages 1240–1247. IEEE.

Guttman, D., Kandemir, M. T., Arunachalamy, M., and Calina, V. (2015). Performance and energy evaluation of data prefetching on Intel Xeon Phi. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 288–297. IEEE.

Han, S., Jang, K., Park, K., and Moon, S. (2010). PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206.

Hennessy, J. L. and Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach*. Elsevier.

Herlihy, M., Shavit, N., and Tzafrir, M. (2008). Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer.

Huang, K., Xie, G., Li, Y., and Liu, A. X. (2011). Offset addressing approach to memory-efficient IP address lookup. In *Proceedings of IEEE INFOCOM*, pages 306–310.

Intel (2005). Intel® I/O acceleration technology. `https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html`.

Intel (2009). An Introduction to the Intel® QuickPath Interconnect. `https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html`.

Intel (2012). Intel® Data Direct I/O Technology. `https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html`.

Intel (2015). Intel® Ethernet Flow Director. `https://connectedsocialmedia.com/13463/intel-ethernet-flow-director-fortville-video/`.

Intel (2019a). Intel® 64 and IA-32 Architectures Optimization Reference Manual. `https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual`.

Intel (2019b). Intel® 82599 10 GbE Controller Datasheet. `http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html`.

Intel (2019c). Intel® Vtune Profiler. `https://software.intel.com/en-us/vtune/documentation/get-started`.

IO Visor Project (2015). BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. `https://github.com/iovisor/bcc`.

ISO/IEC (2017). Technical Specification — C++ Extensions for Coroutines. `https://www.iso.org/standard/73008.html`.

Jain, P. (2016). Recent trends in next generation terabit Ethernet and gigabit wireless local area network. In *2016 International Conference on Signal Processing and Communication (ICSC)*, pages 106–110. IEEE.

Jonathan, C., Minhas, U. F., Hunter, J., Levandoski, J., and Nishanov, G. (2018). Exploiting coroutines to attack the killer nanoseconds. *Proceedings of the VLDB Endowment*, 11(11):1702–1714.

Joseph, D. and Grunwald, D. (1999). Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133.

Kalia, A., Zhou, D., Kaminsky, M., and Andersen, D. G. (2015). Raising the bar for using GPUs in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 409–423.

Kang, M. S., Lee, S. B., and Gligor, V. D. (2013). The crossfire attack. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 127–141. IEEE.

Klaiber, A. C. and Levy, H. M. (1991). An architecture for software-controlled data prefetching. *ACM SIGARCH Computer Architecture News*, 19(3):43–53.

Knuth, D. E. (1973). *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley.

Knuth, D. E. (1997). *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley Professional.

Kocberber, O., Falsafi, B., and Grot, B. (2015). Asynchronous memory access chaining. *Proceedings of the VLDB Endowment*, 9(4):252–263.

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297.

Koponen, T., Amidon, K., Balland, P., Casado, M., Chanda, A., Fulton, B., Ganichev, I., Gross, J., Ingram, P., Jackson, E., et al. (2014). Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216.

Lameter, C. (2013). NUMA (non-uniform memory access): An overview. *Queue*, 11(7):40–51.

Le, H., Jiang, W., and Prasanna, V. K. (2008). A SRAM-based Architecture for Trie-based IP Lookup Using FPGA. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 33–42. IEEE.

Le, H. and Prasanna, V. K. (2011). Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning. *IEEE Transactions on Computers*, 61(7):1026–1039.

Le, Y., Chang, H., Mukherjee, S., Wang, L., Akella, A., Swift, M. M., and Lakshman, T. (2017). UNO: uniflying host and smart NIC offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519.

Lee, J., Kim, H., and Vuduc, R. (2012). When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(1):1–29.

Lehmann, M. (2018). libcoro: C-library that implements coroutines (cooperative multi-tasking) in a portable fashion. `http://software.schmorp.de/pkg/libcoro.html`.

Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. (2017). KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152.

Li, B., Tan, K., Luo, L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. (2016). ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14.

Li, P. and Luo, Y. (2016). P4GPU: Accelerate packet processing of a P4 program with a CPU-GPU heterogeneous architecture. In *2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 125–126.

Li, S., Lim, H., Lee, V. W., Ahn, J. H., Kalia, A., Kaminsky, M., Andersen, D. G., Seongil, O., Lee, S., and Dubey, P. (2015). Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 476–488.

Liao, S.-w., Hung, T.-H., Nguyen, D., Chou, C., Tu, C., and Zhou, H. (2009). Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–10.

Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. (2014). MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444.

Lim, H. and Lee, N. (2011). Survey and proposal on binary search algorithms for longest prefix match. *IEEE Communications Surveys & Tutorials*, 14(3):681–697.

Lim, H., Lim, K., Lee, N., and Park, K.-H. (2012). On adding Bloom filters to longest prefix matching algorithms. *IEEE Transactions on Computers*, 63(2):411–423.

Lim, H., Yim, C., and Swartzlander, E. E. (2010). Priority tries for IP address lookup. *IEEE Transactions on Computers*, 59(6):784–794.

Linux (2009). perf. `https://github.com/torvalds/linux/tree/master/tools/perf`.

Liu, M., Cui, T., Schuh, H., Krishnamurthy, A., Peter, S., and Gupta, K. (2019a). Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the 2019 ACM Conference on SIGCOMM*, pages 318–333.

Liu, M., Peter, S., Krishnamurthy, A., and Phothilimthana, P. M. (2019b). E3: energy-efficient microservices on SmartNIC-accelerated servers. In *2019 USENIX Annual Technical Conference (ATC 19)*, pages 363–378.

Liu, X., Yang, X., and Lu, Y. (2008). To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. In *Proceedings of the 2008 ACM conference on SIGCOMM*, pages 195–206.

Liu, Z., Ben-Basat, R., Einziger, G., Kassner, Y., Braverman, V., Friedman, R., and Sekar, V. (2019c). Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *Proceedings of the 2019 ACM conference on SIGCOMM*, page 334–350, New York, NY, USA. Association for Computing Machinery.

Luo, L., Xie, G., Salamatian, K., Uhlig, S., Mathy, L., and Xie, Y. (2013). A trie merging approach with incremental updates for virtual routers. In *Proceedings of IEEE INFOCOM*, pages 1222–1230. IEEE.

Machado, M., Doucette, C., Fu, Q., and Byers, J. (2020). Gatekeeper: the first open-source DDoS protection system. `https://github.com/AltraMayor/gatekeeper`.

Mahajan, R., Bellovin, S. M., Floyd, S., Ioannidis, J., Paxson, V., and Shenker, S. (2002). Controlling high bandwidth aggregates in the network. *ACM SIGCOMM Computer Communication Review*, 32(3):62–73.

Makrushin, D. (2017). The cost of launching a DDoS attack. `https://securelist.com/analysis/publications/77784/the-cost-of-launching-a-ddos-attack/`.

Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. (2014). ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473.

Marvell (2020). OCTEON TX2 64-bit ARM-based SoCs. `https://www.marvell.com/products/infrastructure-processors/multi-core-processors/octeon-tx2.html`.

McCanne, S. and Jacobson, V. (1993). The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter*, volume 46.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.

Mehta, S., Fang, Z., Zhai, A., and Yew, P.-C. (2014). Multi-stage coordinated prefetching for present-day processors. In *Proceedings of the 2014 International Conference on Supercomputing*, pages 73–82.

Mellanox (2020). Mellanox BlueField SmartNIC. `https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf`.

Meribout, M. and Motomura, M. (2003). A new hardware algorithm for fast IP routing targeting programmable routers. In *International Conference on Network Control and Engineering for QoS, Security and Mobility*, pages 164–179. Springer.

Miano, S., Doriguzzi-Corin, R., Risso, F., Siracusa, D., and Sommese, R. (2019). Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case. *IEEE Access*, 7:107161–107170.

Miao, R., Potharaju, R., Yu, M., and Jain, N. (2015). The dark menace: Characterizing network-based attacks in the cloud. In *Proceedings of the 2015 Internet Measurement Conference*, pages 169–182.

Miao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. (2017). SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the 2017 ACM conference on SIGCOMM*, pages 15–28. ACM.

Mittal, M. (2010). Deterministic lookup using hashed key in a multi-stride compressed trie structure. US Patent 7,827,218.

Mittal, S. (2016). A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, 49(2):1–35.

Moon, Y., Lee, S., Jamshed, M. A., and Park, K. (2020). AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92.

Moura, A. L. D. and Ierusalimschy, R. (2009). Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31.

Mowry, T. and Gupta, A. (1991). Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106.

Mowry, T. C. et al. (1994). *Tolerating latency through software-controlled data prefetching*. PhD thesis, to the Department of Electrical Engineering.Stanford University.

Nesbit, K. J., Dhodapkar, A. S., and Smith, J. E. (2004). AC/DC: An adaptive data cache prefetcher. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 135–145. IEEE.

Nesbit, K. J. and Smith, J. E. (2004). Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 96–96. IEEE.

Netronome (2018). Netronome Agilio LX 1x100GbE SmartNIC. `https://www.netronome.com/m/documents/PB_Agilio_Lx_1x100GbE.pdf`.

NETSCOUT (2019). 14th Annual Worldwide Infrastructure Security Report. `https://www.netscout.com/report/`.

NETSCOUT (2019). Mirai's Botnet Tsunami. `https://www.netscout.com/blog/mirais-botnet-tsunami`.

Nilsson, S. and Karlsson, G. (1999). IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092.

ntop (2019). PF_RING: high-speed packet capture, filtering and analysis. `https://www.ntop.org/products/packet-capture/pf_ring/`.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113. Wiley Online Library.

Pagh, R. and Rodler, F. F. (2001). Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer.

Panigrahy, R. and Sharma, S. (2002). Reducing TCAM power consumption and increasing throughput. In *Proceedings of the 10th Symposium on High Performance Interconnects*, pages 107–112. IEEE.

Pao, D., Lu, Z., and Poon, Y. H. (2014). IP address lookup using bit-shuffled trie. *Computer Communications*, 47:51–64.

Parno, B., Wendlandt, D., Shi, E., Perrig, A., Maggs, B., and Hu, Y.-C. (2007). Portcullis: Protecting connection setup from Denial-of-Capability attacks. *ACM SIGCOMM Computer Communication Review*, 37(4):289–300.

Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. (2015). The design and implementation of open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130.

Phothilimthana, P. M., Liu, M., Kaufmann, A., Peter, S., Bodik, R., and Anderson, T. (2018). Floem: a programming system for NIC-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679.

Porterfield, A. K. (1989). *Software methods for improvement of cache performance on supercomputer applications*. PhD thesis, Rice University.

Psaropoulos, G., Legler, T., May, N., and Ailamaki, A. (2017). Interleaving with coroutines: a practical approach for robust index joins. *Proceedings of the VLDB Endowment*, 11(CONF):230–242.

Psaropoulos, G., Legler, T., May, N., and Ailamaki, A. (2019). Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *The VLDB Journal*, 28(4):451–471.

Quagga (2018). Quagga Routing Suite. `http://www.nongnu.org/quagga/`.

Rahman, S., Burtscher, M., Zong, Z., and Qasem, A. (2015). Maximizing hardware prefetch effectiveness with machine learning. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC'15)*, pages 383–389. IEEE.

Rétvári, G., Tapolcai, J., Kőrösi, A., Majdán, A., and Heszberger, Z. (2013). Compressing IP forwarding tables: Towards entropy bounds and beyond. *ACM SIGCOMM Computer Communication Review*, 43(4):111–122.

RIPE Network Coordination Centre (2017). RIS Raw Data. `https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data`.

Rizzo, L. (2012). Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (Security 12)*, pages 101–112.

Ruiz-Sánchez, M. Á., Biersack, E. W., and Dabbous, W. (2001). Survey and taxonomy of IP address lookup algorithms. *IEEE Network*, 15(2):8–23.

Sahni, S. and Lu, H. (2007). Dynamic tree bitmap for IP lookup and update. In *Sixth International Conference on Networking (ICN'07)*, pages 79–79. IEEE.

Sangireddy, R., Futamura, N., Aluru, S., and Somani, A. K. (2005). Scalable, memory efficient, high-speed IP lookup algorithms. *IEEE/ACM Transactions on Networking*, 13(4):802–812.

Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. (2012). Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 323–336.

Shah, D. and Gupta, P. (2001). Fast updating algorithms for TCAM. *IEEE Micro*, 21(1):36–47.

Sherry, J., Gao, P. X., Basu, S., Panda, A., Krishnamurthy, A., Maciocco, C., Manesh, M., Martins, J., Ratnasamy, S., Rizzo, L., et al. (2015). Rollback-recovery for middleboxes. In *Proceedings of the 2015 ACM Conference on SIGCOMM*, pages 227–240.

Siracusano, G. and Bifulco, R. (2017). Is it a SmartNIC or a Key-Value Store? Both! In *Proceedings of the SIGCOMM Posters and Demos*, pages 138–140.

Sklower, K. (1991). A Tree-Based Packet Routing Table for Berkeley Unix. In *USENIX Winter*, volume 1991, pages 93–99.

Smith, A. J. (1982). Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530.

Snavely, A., Carter, L., Boisseau, J., Majumdar, A., Gatlin, K. S., Mitchell, N., Feo, J., and Koblenz, B. (1998). Multi-processor performance on the Tera MTA. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 4–4. IEEE.

Somogyi, S., Wenisch, T. F., Ailamaki, A., Falsafi, B., and Moshovos, A. (2006). Spatial memory streaming. *ACM SIGARCH Computer Architecture News*, 34(2):252–263.

Song, H., Kodialam, M., Hao, F., and Lakshman, T. (2009). Scalable IP lookups using shape graphs. In *17th IEEE International Conference on Network Protocols (ICNP 09)*, pages 73–82.

Song, H., Kodialam, M., Hao, F., and Lakshman, T. (2010). Building scalable virtual routers with trie braiding. In *Proceedings of IEEE INFOCOM*, pages 1–9. IEEE.

Srinivasan, V. and Varghese, G. (1999). Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40.

Tilera (2014). Tilera Datasheet. `http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8036_PB033-02_web.pdf`.

Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., and Stamm, R. L. (1996). Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202.

Tullsen, D. M., Eggers, S. J., and Levy, H. M. (1995). Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403.

VMware (2012). VMware vSphere Distributed Switch Best Practices white paper. `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vsphere-distributed-switch-best-practices-white-paper.pdf`.

Waldvogel, M., Varghese, G., Turner, J., and Plattner, B. (1997). Scalable high speed IP routing lookups. In *Proceedings of the 1997 ACM conference on SIGCOMM*, pages 25–36.

Wang, F. and Hamdi, M. (2008). Matching the Speed Gap between SRAM and DRAM. In *2008 International Conference on High Performance Switching and Routing*, pages 104–109. IEEE.

Wang, H., Soulé, R., Dang, H. T., Lee, K. S., Shrivastav, V., Foster, N., and Weatherspoon, H. (2017). P4FGPA: A rapid prototyping framework for P4. In *Proceedings of the Symposium on SDN Research*, pages 122–135.

Wang, M., Deering, S., Hain, T., and Dunn, L. (2004). Non-random generator for IPv6 tables. In *Proceedings of the 12th Annual IEEE Symposium on High Performance Interconnects*, pages 35–40.

Wang, Z., Burger, D., McKinley, K. S., Reinhardt, S. K., and Weems, C. C. (2003). Guided region prefetching: a cooperative hardware/software approach. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 388–398. IEEE.

Warkhede, P., Suri, S., and Varghese, G. (2004). Multiway range trees: scalable IP lookup with fast updates. *Computer Networks*, 44(3):289–303.

Xie, K., Li, X., Wang, X., Cao, J., Xie, G., Wen, J., Zhang, D., and Qin, Z. (2018a). Online anomaly detection with high accuracy. *IEEE/ACM Transactions on Networking*, 26(3):1222–1235.

Xie, K., Li, X., Wang, X., Xie, G., Wen, J., Cao, J., and Zhang, D. (2017). Fast Tensor Factorization for Accurate Internet Anomaly Detection. *IEEE/ACM Transactions on Networking*, 25(6):3794–3807.

Xie, K., Wang, L., Wang, X., Xie, G., Wen, J., Zhang, G., Cao, J., and Zhang, D. (2018b). Accurate recovery of Internet traffic data: A sequential tensor completion approach. *IEEE/ACM Transactions on Networking*, 26(2):793–806.

Xilinx (2018). 7 Series FPGAs Data Sheet: Overview. `http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`.

Yaar, A., Perrig, A., and Song, D. (2004). SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 130–143. IEEE.

Yang, T. (2014). SAIL Webpage. `http://net.pku.edu.cn/~yangtong/pages/SAIL.html`.

Yang, T., Duan, R., Lu, J., Zhang, S., Dai, H., and Liu, B. (2012a). Clue: Achieving fast update over compressed table for parallel lookup with reduced dynamic redundancy. In *IEEE 32nd International Conference on Distributed Computing Systems*, pages 678–687.

Yang, T., Liu, A. X., Shahzad, M., Yang, D., Fu, Q., Xie, G., and Li, X. (2017). A shifting framework for set queries. *IEEE/ACM Transactions on Networking*, 25(5):3116–3131.

Yang, T., Liu, A. X., Shahzad, M., Zhong, Y., Fu, Q., Li, Z., Xie, G., and Li, X. (2016). A shifting Bloom filter framework for set queries. *Proceedings of the VLDB Endowment*, 9(5):408–419.

Yang, T., Liu, A. X., Shen, Y., Fu, Q., Li, D., and Li, X. (2018a). Fast OpenFlow table lookup with fast update. In *Proceedings of IEEE INFOCOM*, pages 2636–2644. IEEE.

Yang, T., Mi, Z., Duan, R., Guo, X., Lu, J., Zhang, S., Sun, X., and Liu, B. (2012b). An ultra-fast universal incremental update algorithm for trie-based routing lookup. In *20th IEEE International Conference on Network Protocols (ICNP 12)*, pages 1–10. IEEE.

Yang, T., Xie, G., Li, Y., Fu, Q., Liu, A. X., Li, Q., and Mathy, L. (2014). Guarantee IP lookup performance with FIB explosion. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 39–50.

Yang, T., Xie, G., Liu, A. X., Fu, Q., Li, Y., Li, X., and Mathy, L. (2018b). Constant IP lookup with FIB explosion. *IEEE/ACM Transactions on Networking*, 26(4):1821–1836.

Yang, X., Wetherall, D., and Anderson, T. (2005). A DoS-limiting network architecture. *ACM SIGCOMM Computer Communication Review*, 35(4):241–252.

Yu, M., Fabrikant, A., and Rexford, J. (2009). BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proceedings of the 5th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 09)*, pages 313–324.

Zane, F., Narlikar, G., and Basu, A. (2003). CoolCAMs: Power-efficient TCAMs for forwarding engines. In *Proceedings of IEEE INFOCOM*, volume 1, pages 42–52. IEEE.

Zec, M., Rizzo, L., and Mikuc, M. (2012). DXR: towards a billion routing lookups per second in software. *ACM SIGCOMM Computer Communication Review*, 42(5):29–36.

Zhao, J., Zhang, X., Wang, X., and Xue, X. (2010). Achieving O(1) IP lookup on GPU-based software routers. In *Proceedings of the 2010 ACM conference on SIGCOMM*, pages 429–430.

Zheng, K., Hu, C., Lu, H., and Liu, B. (2006). A TCAM-based distributed parallel IP lookup scheme and performance analysis. *IEEE/ACM Transactions on Networking*, 14(4):863–875.

Zhou, D., Fan, B., Lim, H., Kaminsky, M., and Andersen, D. G. (2013). Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 13)*, pages 97–108.

Zucker, D. F., Lee, R. B., and Flynn, M. J. (2000). Hardware and software cache prefetching techniques for MPEG benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):782–796.

CURRICULUM VITAE

Qiaobin Fu

Address	111 Cummington Mall Department of Computer Science, Boston University Boston, MA 02215
Email	qiaobinf@bu.edu
Website	http://cs-people.bu.edu/qiaobinf/
Education	**PhD** in Computer Science · Boston University, USA, 2020 · Advisor: Prof. John W. Byers **M.S.** in Computer Science · University of Chinese Academy of Sciences, 2014 · Advisor: Prof. Gaogang Xie **B.E.** in Software Engineering · Dalian University of Technology (DUT), China, 2011
Publications	Michel Machado, Cody Doucette, **Qiaobin Fu**, John W. Byers. The design and deployment of a DDoS protection system. Working paper targeted at USENIX NSDI, Boston, MA, USA, April 2021. Tong Yang, **Qiaobin Fu**, Alex X. Liu, Kun Huang, Gong Zhang, Bin Cui, Steve Uhlig. Fit the Elephant in a Box - Towards IP Lookup at On-chip Memory Access Speed. Submitted to USENIX ATC, Boston, MA, USA, July 2020. Yibo Yan, **Qiaobin Fu**, Yang Zhou, Alex X. Liu, Tong Yang, Bin Cui, Tengjiao Wang. On the Controversy of Bloom Filter False Positives - An Information Theoretical Approach to Optimizing Bloom Filter Parameters. Submitted to USENIX ATC, Boston, MA, USA, July 2020. Yinda Zhang, Peiqing Chen, Duo Zhang, **Qiaobin Fu**, Tong Yang, Alex X. Liu, Gong Zhang, Bin Cui and Steve Uhlig. Cuckoo-Group: A Flexible and Weight-Aware Cuckoo Filter. Submitted to ACM SIGKDD, San Diego, California, USA, August 2020.

Tong Yang, Gaogang Xie, Alex X. Liu, **Qiaobin Fu**, Yanbiao Li, Xiaoming Li, Laurent Mathy. “Constant IP Lookup with FIB Explosion.” In IEEE/ACM ToN, 2018.

Qiaobin Fu, Nicholas Timkovich, Pierre Riteau, Kate Keahey. “A Step towards Hadoop Dynamic Scaling.” In IEEE HPCC 2018.

Tong Yang, Alex X. Liu, Yulong Shen, **Qiaobin Fu**, Dagang Li, Xiaoming Li. “Fast OpenFlow Table Lookup with Fast Update.” In IEEE INFOCOM 2018.

Tong Yang, Alex Liu, Muhammad Shahzad, Dongsheng Yang, **Qiaobin Fu**, Gaogang Xie, Xiaoming Li. “A Shifting Framework for Set Queries.” In IEEE/ACM ToN, 2017.

Tong Yang, Alex X. Liu, Muhammad Shahzad, Yuankun Zhong, **Qiaobin Fu**, Zi Li, Gaogang Xie, Xiaoming Li. “A Shifting Bloom Filter Framework for Set Queries.” In Proceedings of the 42th International Conference on Very Large Data Bases (VLDB), Vol. 9, New Delhi, India, September, 2016.

Tong Yang, Alex X. Liu, **Qiaobin Fu**, Dongsheng Yang, Steve Uhlig, Xiaoming Li. “Fit the Elephant in a Box - Towards IP Lookup at On-chip Memory Access Speed.” In IEEE ICNP 2016, Poster.

Tong Yang, Gaogang Xie, Yanbiao Li, **Qiaobin Fu**, Alex X. Liu, Qi Li, Laurent Mathy. “Guarantee IP Lookup Performance with FIB Explosion.” In Proceedings of the ACM SIGCOMM Conference (SIGCOMM), Chicago, Illinois, August, 2014.

Teaching Experience

Mentor for Google Summer of Code (GSoC), 2015 – 2018.

Teaching Fellow for CS 350: Fundamentals of Computing Systems.
Working with Prof. Azer Bestavros (Fall 2018, Spring 2018, Fall 2016, Spring 2016, and Spring 2015), and Prof. Renato Mancuso (Fall 2018, and Spring 2018).

Teaching Fellow for CS 455/655: Introduction to Computer Networks.
Working with Prof. Abraham Matta (Fall 2017).