ISLAM, MD I., August 2022 COMPUTER SCIENCE AN IPV6 ROUTING TABLE LOOKUP ALGORITHM IN SOFTWARE AND ASIC BY DESIGN- ING A HIGH-LEVEL SYNTHESIS SYSTEM (114 pages)

Dissertation Advisor: Javed I. Khan

This dissertation has two contributions. The primary contribution is to present a trie based rout- ing table lookup algorithm named CP-Trie. The secondary contribution is to present a high-level synthesis tool named C2RTL that can generate routing table lookup implementation in ASIC from C code.

Routing table lookup is a key function of a router. It involves performing the longest prefix match (LPM). A router needs to perform a routing table lookup for each incoming packet. High- speed routers generally implement routing table lookup in Software and ASIC (Application Specific Integrated Circuit). This dissertation describes a new routing table algorithm named CP-Trie that outperforms the state-of-the-art trie based routing table lookup algorithm in lookup speed while consuming slightly more memory. We evaluated our algorithms with real routing tables from RouteView project. Our experiments with real routing tables from core routers show that CP- Trie achieves upto 1.43X lookup throughput on a general purpose CPU, but consumes 1.36-1.47X memory compared to the state-of-the-art solution. CP-Trie also outperforms the state-of-the-art solutions in ASIC.

Implementing routing table lookup in ASIC is another challenge. The ASICs in high-speed routers are currently designed in a register transfer level (RTL) hardware description language (HDL) such as Verilog or VHDL. However, manually writing hardware logic is notoriously com- plicated and painful. This dissertation describes a high-level synthesis (HLS) tool named C2RTL that can generate Verilog RTL from C code. It takes a routing table lookup algorithm in C as an input and generates corresponding Verilog RTL code. We used C2RTL to generate the Verilog RTL implementation of CP-Trie. We then synthesized the generated RTL code with OpenROAD in a 1 GHz pipelined ASIC with a 45nm standard cell library. Our OpenROAD report shows that CP-Trie consumes 14% less power and 20.5% less area compared to the state-of-the-art solution in ASIC.

# AN IPV6 ROUTING TABLE LOOKUP ALGORITHM IN SOFTWARE AND ASIC BY DESIGNING A HIGH-LEVEL SYNTHESIS SYSTEM

A dissertation submitted to Kent State University

in partial fulfillment of the requirements for the degree of Doctor of Philosophy

by

MD Iftakharul Islam August 2022

© Copyright All rights reserved

Except for previously published materials

Dissertation written by MD Iftakharul Islam

PhD in Computer Science, Kent State University, 2022

Approved by

Javed I. Khan , Chair, Doctoral Dissertation Committee

Hassan Peyravi , Members, Doctoral Dissertation Committee

Jong-Hoon Kim

Murali Shanker

Accepted by

Javed I. Khan , Chair, Department of Computer Science

Mandy Munro-Stasiuk , Dean, College of Arts and Sciences

# TABLE OF CONTENTS

# TABLE OF CONTENTS iv

# LIST OF FIGURES viii

# LIST OF TABLES x

# ACKNOWLEDGMENTS xi

# Introduction 1

|  |  |  |
| --- | --- | --- |
| 1.1 | Throughput and Scalability Requirement . . . . . . . . . . . . . . . . . . . . . . . . . | 2 |
| 1.2 | Trie based IP lookup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 3 |
| 1.3 | Challenges of ASIC Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 4 |
| 1.4 | Proposed Solutions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 5 |
|  | 1.4.1 CP-Trie . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 5 |
|  | 1.4.2 C2RTL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 6 |
| 1.5 | Source code Availability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 6 |
| 1.6 | Evaluation metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 7 |
| 1.7 | Organization of the dissertation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 7 |
| **2 An** | **overview of a router . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .** | **9** |
| 2.1 | Convergence Between Routers and Switches . . . . . . . . . . . . . . . . . . . . . . . | 10 |
| 2.2 | Router Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 11 |
| 2.3 | An overview of router forwarding plane . . . . . . . . . . . . . . . . . . . . . . . . . . | 12 |
|  | 2.3.1 Ingress Packet Processor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 14 |
| 2.4 | Multiple pipelines in a routing chip . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 17 |
| **3 An** | **overview of Trie data structure . . . . . . . . . . . . . . . . . . . . . . . . . .** | **18** |
|  | 3.0.1 Encoding Trie with bitmaps . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 19 |

1. **Related works on Trie based IP lookup algorithms and implementations . . . 22**

[4.0.1 Trie based IP lookup implementations 23](#_TOC_250047)

1. Related works on high-level synthesis 24

[5.0.1 High-level Synthesis for IP Lookup 26](#_TOC_250046)

1. CP-Trie: Cumulative PopCount based Trie for IPv6 Routing Table Lookup in Software and ASIC 27
   1. [Proposed Approach 28](#_TOC_250045)
   2. [Lookup Algorithm 31](#_TOC_250044)
   3. [Implementation 33](#_TOC_250043)
      1. [Software Implementation 33](#_TOC_250042)
      2. [ASIC Implementation 33](#_TOC_250041)
   4. [Evaluation of ASIC 34](#_TOC_250040)
   5. [EXPERIMENTAL RESULTS 35](#_TOC_250039)
      1. [Data Set 36](#_TOC_250038)
      2. [Memory Consumption 36](#_TOC_250037)
      3. [Traffics for FIB Lookup 37](#_TOC_250036)
      4. [Lookup Performance in Software 38](#_TOC_250035)
      5. [Update Performance in Software 40](#_TOC_250034)
   6. [Discussion 40](#_TOC_250033)
   7. [Conclusion 41](#_TOC_250032)
2. C2RTL: A High-level Synthesis System for IP Lookup 42
   1. [IP lookup and Packet classification Algorithms in ASIC 43](#_TOC_250031)
   2. [Programming Convention 45](#_TOC_250030)
   3. [System Overview 45](#_TOC_250029)
      1. [Resource Binding 50](#_TOC_250028)
   4. [MUX Tree Generation 51](#_TOC_250027)
   5. [Evaluations 52](#_TOC_250026)
      1. [Physical Synthesis 53](#_TOC_250025)
   6. [Discussion 54](#_TOC_250024)
   7. [Conclusion 55](#_TOC_250023)
3. Using Cumulative PopCount to improve SAIL based IPv4 Lookup in Linux Kernel and Domino based programmable ASIC 56
   1. [FIB Lookup in IPv4 56](#_TOC_250022)
   2. [Population Counting 59](#_TOC_250021)
   3. [Lookup Algorithm 62](#_TOC_250020)
   4. [Implementation 64](#_TOC_250019)
      1. [SAIL in a Programmable Pipeline 65](#_TOC_250018)
   5. [Comparison between SAIL U and SAIL L in a Programmable Pipeline 66](#_TOC_250017)
   6. [EXPERIMENTAL RESULTS 67](#_TOC_250016)
      1. [Data Set 67](#_TOC_250015)
      2. [Experimental Setup 68](#_TOC_250014)
      3. [Memory Consumption 68](#_TOC_250013)
      4. [Lookup Cost 69](#_TOC_250012)
      5. [Update Cost 71](#_TOC_250011)
   7. [Conclusion 71](#_TOC_250010)
4. Limitations 73
   * 1. [Limitations common to both CP-Trie and C2RTL project 73](#_TOC_250009)
     2. [Limitations of CP-Trie 75](#_TOC_250008)
     3. [Limitations of C2RTL 75](#_TOC_250007)
     4. [Future work on Power/Area Efficient Pipelining 77](#_TOC_250006)
     5. [Future work on broader data-plane function support 83](#_TOC_250005)
5. Conclusion 85
   * 1. [Broader Impact of CP-Trie 85](#_TOC_250004)
     2. [Broader Impact of C2RTL 86](#_TOC_250003)
     3. [Future work on CP-Trie 87](#_TOC_250002)
     4. [Future work on C2RTL 90](#_TOC_250001)

[BIBLIOGRAPHY 97](#_TOC_250000)

# LIST OF FIGURES

1. IP lookup implementations 3
2. Architecture of the Internet 10
3. Architecture of a Router 11
4. Forwarding plane of a router. 12
5. Modules of an ingress packet processor. 14
6. Packet headers. 14
7. An example Trie for a dictionary consisting of the words ”am”, ”an”, ”and”, ”ant”, ”ants”, ”do”, ”dog”, ”dogs”, ”don”, ”done”, ”dons”, ”so”, ”sad”, ”sick” 19
8. Trie based IP lookup 22
9. CP-Trie data structure 29
10. Index calculation for the next level (*B*6) and *Leaf* 29
11. Memory consumption for different FIBs 36
12. Lookup throughputs in different FIBs for different traffics 39
13. Average insertion time per prefix (smaller is better) 40
14. An example trie based IP lookup in C 43
15. C2RTL design flow 46
16. Intermediate code generation by GCC 47
17. CDFG corresponding to Figure 17b 48
18. Scheduling 49
19. MUX tree generation 52
20. C code to physical chip layout generation 53
21. Programmable pipeline based Linux router 58
22. SAIL U with *population counting* 59
23. Memory consumption for different FIBs 69
24. Lookup cost for different levels 70
25. Update cost for different prefix lengths 71
26. Block diagram of a pipelined ASIC 77
27. Example pipeline with Initiation Interval (II) = 1 and Latency = 3 cycle 78
28. Four arbitrary pipelining options of the same circuit (CDFG) 79

# LIST OF TABLES

1. Example IPv6 routing table 1
2. Source code, data, scripts, slides and presentations for CP-Trie and C2RTL projects 6
3. Example ruleset (for packet classification) 15
4. Example forwarding table (for IP lookup) 16
5. Power, area and timing in ASIC (for logic) 34
6. FIB dataset 35
7. programming restrictions in C2RTL 44
8. An example predicates for a SSA *φ* operation 51
9. Power, area and timing in ASIC 54
10. Routing table (also known as FIB table) 57
11. Data structures used in SAIL 62
12. Comparison between SAIL U and SAIL L in a Programmable Pipeline 66
13. FIB dataset 67
14. Impact of population counting on memory consumption (for *fib6* ) 68
15. Area and delay trade-offs for an adder 76
16. Available resources to implement *adder*, *multiplier*, *left shifter*, *comparator* and *mul- tiplexer* 80
17. Resource allocation options where path latency is less than 1 ns (latency requirement)

for stage 1 in case 1 in Figure 29 81

1. Optimal resource allocation for case 1 in Figure 29. Here, the path latency of each

stage is less than 1 ns, thus latency requirement is met and area in minimized. 81

1. Optimal resource allocation for different pipeline options 82
2. Router data plane functions in C2RTL 84

# ACKNOWLEDGMENTS

Last nine years in grad school was a long, but fun journey towards developing specialized knowledge and skills in network algorithms and high-level synthesis. It was an immense journey for my personal growth as well. Many people were responsible for making it an enjoyable ride, and I would like to thank them.

Firstly, I would like to thank my advisor, Javed Khan. He gave me the freedom to go after the ideas that I thought were interesting. He also encouraged me to pursue very fundamental problems. He always believed in me and allowed me to work independently. It also made my research work productive and fun. He also helped me when I needed any help. I am grateful for his support.

I would also like to thank Fabrizio Ferrandi for helping me to understand high-level synthesis. He is a very knowledgeable, accessible and dynamic person. He also helped me to learn how to use OpenROAD.

I would like to thank my dissertation committee member, Hassan Peyravi. I took his Advanced Networking course which was one of the best courses I have ever taken. There, I learned intricate details of networking. He is an excellent teacher. I learned a lot about teaching from him.

I would like to thank another dissertation committee member, Jong-Hoon Kim. He is always very eager to discuss new research problems which I found delightful.

I would like to thank Mikhail Nesterenko for having me as his TA. I had enormous fun while teaching CS III labs (later the course). I learned a great deal from the course too. I found those very useful while working on my Ph.D. projects. I also learned a lot about course management from him.

I would also like to thank Marcy for helping me throughout my journey since I was in Bangladesh.

She always helped me if I needed any administrative help.

I spent some of the best parts of my life in Kent, OH. I used to go to Kent State library and coffee shops literally every day. I used to walk around the campus, hiking trails and downtown. They were my happy places. I did almost all of my research work at Kent State library and Kent Starbucks. I would like to thank all the great people at Kent State library and Kent Starbucks.

I made some good friends while I was in Kent. They made my life much enjoyable. Their love and friendship gave me the energy to go on in life. I also spent two summers in Bloomington, IN and Columbus, OH for summer internship. I met some wonderful people there as well. Specially, I would like to thank my friends Rob and Andrew for their friendship.

I would also like to thank my friend Rafiq who was with me from the beginning of my grad school journey. We always stayed very close throughout my Ph.D. He provided me with enormous emotional support throughout my Ph.D. I am grateful for our friendship.

I would also like to thank Amjad and Robin. They were my lab mates in Midianet lab. As my predecessor, Amjad helped me in different stages of my PhD. Sometimes, Robin and I used to hang out at nights and discussed about computer science research which was fun.

I used a lot of open source projects for my research. I am grateful to all the developers who contributed to those projects. Many of them helped me to understand those projects as well. I would like to thank David Ahern, Stephen Hemminger, Antonin Bas, Bob Lantz, Sam Xi, Anirudh Sivaraman and so on.

I used to watch a lot of tech talks on YouTube. I learned a great deal from there. I used to watch Netflix and YouTube for fun after midnight everyday. I thank everyone who made those videos possible.

Finally, I would like to thank Linus Torvalds, Steve Jobs and David Graeber. Their thoughts had a huge impression on me. For example, I liked the idea of Linus Torvalds where he does not set a goal for a project and does everything spontaneously. I found this philosophy very useful when I was working on my routing table lookup and high-level synthesis projects. I would like to thanks many other great minds who’s idea contributed to my personal growth, including, Jim Zemlin, Salman Khan, Don Valentine, Bella DePaulo, Katie Hood, Rutger Bregman, Adam Grant, Cenk Uygur, Ana Kasparian, Ashley Vance, Dan Lyons and so on. I dedicate this dissertation to Linus Torvalds, Steve Jobs and David Graeber.

# CHAPTER 1

**Introduction**

Routing table lookup is a key function of a router. It involves performing the longest prefix match (LPM) of an IP address. A router needs to perform a routing table lookup of the destination IP address for each incoming packet. For instance, Table 10 shows an example routing table of a router. If the destination IP address of an incoming packet is 200a:410:8088:500::300, it matches with both *r*2 and *r*3. As *r*3 has longer prefix than *r*2, the packet should be forwarded to next-hop 3. Again, if the destination IP address is 200a:410:8000:702::0de, it matches with *r*2 and *r*4. As *r*4 has longer prefix, the packet should be forwarded to next-hop 1. It has been reported that IP lookup and forwarding engine account for almost 32% of the power consumption in core routers [171, 93]. This is why, an efficient IP lookup engine has a significant impact on router performance. Note that, the terms ”routing table lookup”, ”IP lookup”, and ”Forwarding Information Base (FIB) lookup” have been used interchangeably in the past two decades [7, 184, 50, 47]. In this dissertation, we also use the terms interchangeably.

Table 1: Example IPv6 routing table

|  |  |  |
| --- | --- | --- |
| Route | Prefix | Next-hop |
| r1 | 200*a* : 4 : 112 :: */*48 | 1 |
| r2 | 200*a* : 410 : 8000 :: */*40 | 2 |
| r3 | 200*a* : 410 : 8080 :: */*44 | 3 |
| r4 | 200*a* : 410 : 8000 : 702 :: */*64 | 1 |
| r5 | 200*a* : 410 : 8000 : 702 :: 0*df /*128 | 2 |

# Throughput and Scalability Requirement

IPv6 routing table lookup in core routers requires very high throughput and high scalability. We illustrate the scalability and throughput requirements with following examples:

* + - **Large routing table.** The number of IPv6 routes in core routers already has exceeded 100K and is growing very rapidly. 1 With the exponential growth of the Internet, this number is expected to grow significantly. Thus, an efficient and scalable solution of IP lookup is needed.
    - **High throughput.** Network link rate already has reached 400 Gb/s and growing expo- nentially. As a result, a routing chip needs to achieve very high throughput as well. A 4*.*8 Tb/s Brodcom Jericho2 chip can forward 2 billion packets per second [34, 10]. Other routing chips such as Juniper Express and Cisco Silicon One also have similar performance profiles. Achieving such a high throughput is very challenging.
    - **Longer Prefixes.** An IPv6 lookup may need to match upto 128 bits while IPv4 may need to match upto 32 bits. Thus, IPv6 lookup may require 4× processing and significantly more memory compared to IPv4 lookup.

To achieve the desired scalability and throughput, high-speed routers implement IPv6 lookup algorithms in ASIC. Note that, it is a very common practice is to implement an algorithm in software before implementing it in ASIC. Implementing in software gives more insight about the performance. A more efficient algorithm is likely to perform better in both software and hardware. This is why, algorithms are generally implemented in software before implementing in ASIC. In this dissertation, we focus on software and pipelined ASIC based IPv6 lookup. Also note that, an ASIC is only used for data plane. The dataplane ASIC cannot set its own memory. A software is needed to set up the memory for ASIC. So, in high-speed routers, routing table lookup is implemented both in Software and ASIC. Here, software is primarily responsible for setting up the ASIC memory. On the other hand, ASIC is responsible for actual routing table lookup.

Routing table lookup also can be implemented in TCAM, FPGA and GPU as shown in Figure

1. However, TCAM is very expensive, area-inefficient and power hungry [5, 94]. It also requires very complex incremental update. This is why, it is not suitable for core routers [94, 15, 135].

1<http://bgp.potaroo.net/>



Software

Pipelined ASIC

TCAM

FPGA

GPU

Hash

Hash

Trie

Trie

IP lookup

implementation

Figure 1: IP lookup implementations

FPGA and GPU are known to be an order of magnitude slower than ASIC, thus avoided in real world implementations [7].

# Trie based IP lookup

IP lookup algorithms in software and ASIC mainly fall into two categories: trie [7, 184, 50, 47, 154, 76, 129, 12, 16, 94, 107, 72, 84] and hash [135, 134, 15, 25, 101, 71, 14, 152, 153] based solutions

(in Figure 1). Here, trie based solutions have several significant benefits over hash-based solutions. First of all, hash-based solutions often suffer from collisions. As a result, hash-based solutions cannot guarantee *O*(1) lookup which has significant negative impact. Hash based solutions also require very large memory compared to trie based solutions. Finally, hash-based solutions often also require very expensive update mechanisms (e.g. Cuckoo hashing) [101]. Trie based IP lookup on the other hand consumes very small memory. It can perform lookup in *O*(1). A trie based solution can be considered as multi-level hashing where all the collisions are avoided. Hash based solutions however are simpler to implement than trie based solutions in ASIC. This is why, a router chip manufacturer may choose both trie or hash. This dissertation mainly focuses on trie based solutions.

A trie based IP lookup algorithm should achieve following design goals:

* + - **Fewer number of steps**: The fewer number of steps results in faster lookup. It also reduces the power consumption in ASIC.
    - **Fewer memory accesses:** Fewer memory accesses is very critical in pipelined ASIC. In pipelined ASIC, each pipeline stage has to access SRAM in parallel. Thus, a pipelined ASIC

requires separate SRAM blocks which can serve the pipeline stages simultaneously. 2 It also has been found that fewer large SRAM blocks are more area efficient than many smaller SRAM blocks [134, 135].3 This is why, reducing the number of memory access reduces the number of SRAM blocks needed in ASIC which results in a smaller area.

* + - **Small memory consumption and fast incremental update:** An IP lookup algorithm should consume small memory and fast incremental update.

# Challenges of ASIC Implementation

ASIC based IP lookup is traditionally designed in a register transfer level (RTL) hardware descrip- tion language (HDL) such as Verilog or VHDL. However, designing hardware at RTL level is a very complex and tedious process. Here, designers need to use low level abstraction to implement the logic. The RTL code also needs to be cycle accurate. Thus, designers need to consider the path latency of the circuit and insert registers manually to implement pipelining. Such a complex and tedious process calls for designing hardware at a higher level. High-level synthesis (HLS) [41] allows us to design a pipelined ASIC in a high-level language such as C or SystemC [65]. Here, behavioral/algorithmic designs in ANSI C/C++/SystemC code is synthesized to RTL. Thus, HLS tools generate RTL level hardware logic directly from programs implemented in high level languages such as C or SystemC. This has major advantages over traditional RTL design such as increased designer productivity, better complexity management, shorter simulation cycle (no hardware/soft- ware codesign is needed), rapid design space exploration, higher quality of results (QoRs) and so on [31, 166, 97]. Despite the benefits, HLS has not been adopted in switching and routing chips. Currently, routing ASICs such as Brodcom Jericho+ [21], Cisco Silicon One [36] and Juniper Ex- press [95] are designed in a RTL-level HDL, to the best of our knowledge [33]. Note that, adopting high-level synthesis has significant advantages over traditional RTL design in routing chips. As developing a new algorithm in ASIC can be very challenging, router manufacturers are still us- ing decades old algorithms in their routers. High-level synthesis will enable us to design higher

2It is also possible to implement pipelined ASIC using a multiport SRAM. However, developing a multiport SRAM requires designing a custom SRAM cell based on the required number of ports [8, 128]. Increasing the number of ports increases the area and power consumption of the SRAM significantly. This is why, most of the real-world pipelined ASICs use separate SRAM blocks instead of a multiport SRAM [134, 135].

3This is because control logics, address decoders and sense amplifiers of SRAM consume around 85% of the SRAM

area [135].

performance and more power efficient routers.

# Proposed Solutions

This dissertation has two contributions. The primary contribution here is to describe a new **routing table lookup named CP-Trie [86]**. The secondary contribution of this dissertation is to present a **high-level synthesis (HLS) tool named C2RTL [85]** that was primarily designed for trie based IP lookup algorithms.

# CP-Trie

CP-Trie is an extension of Poptrie [7]. Both CP-Trie and Poptrie are variants of a trie, hence they all offer *O*(1) lookup speed. The main difference between CP-Trie and Poptrie is quantitative. CP-Trie utilizes a new data structure to reduce the number of levels in a trie. Thus, a trie produced by CP-Trie needs fewer levels than that of a Poptrie. Thus, CP-Trie requires fewer steps in lookup. This is how, CP-Trie improves the lookup speed by reducing the constant factor in a trie. Reducing the number of levels in a trie may seem a simple improvement, but it has significant performance benefits in IP lookup in ASIC (which will be described in Chapter 6). As our solution is an extension of Poptrie, we describe Poptrie before describing CP-Trie.

Poptrie [7], the state-of-the-art trie based routing table lookup, encodes nodes using population counting bitmap [176]. Poptrie uses PopCount CPU instruction which can process only 64 bits at a time. This is why, Poptrie uses 6-bit stride (26 = 64). This dissertation presents an extension of Poptrie named CP-Trie (stands for Cumulative PopCount based Trie) where it stores cumulative PopCount along with population counting bitmap. This enables CP-Trie to process longer stride (e.g. 8 − 16 bits) at each step. This reduces the number of steps and the number of memory access

needed for an IP lookup. The fewer number of steps results in faster lookup. It also results in less

power consumption in ASIC. Fewer memory accesses indicate that it requires fewer SRAM blocks in ASIC which results in lower area. This is why, CP-Trie is a more practical solution for pipelined ASIC compared to Poptrie. Our experiments with routing tables from real core routers show that CP-Trie achieves upto 1*.*43× lookup throughput on a general purpose CPU, but consumes

1*.*36 − 1*.*47× memory compared to Poptrie. We also implemented Poptrie and CP-Trie in a 1

GHz pipelined ASIC. Our physical synthesis report shows that CP-Trie consumes 0*.*86× power and 0*.*79× area compared to Poptrie in ASIC.

# C2RTL

C2RTL generates hardware logic in Verilog RTL directly from IP lookup algorithm implemented in

C. C2RTL is implemented as a plugin of GCC 4 compiler. It takes an IP lookup algorithm (in C) as an input and generates corresponding synthesizable Verilog RTL code for pipelined ASIC. We developed several IP lookup algorithms such as CP-Trie [86], Poptrie [7], and SAIL [184] in C2RTL and generated corresponding Verilog RTL. We evaluated the resulting RTL code with OpenROAD.

Table 2: Source code, data, scripts, slides and presentations for CP-Trie and C2RTL projects

|  |  |  |
| --- | --- | --- |
|  | URL | Content |
| CP-Trie | https://tamimcse.github.io/cp-trie | Source code Routing table data Traffic data  Performance report |
| C2RTL | https://tamimcse.github.io/c2rtl | Source code  IP lookup algorithms implementation Packet classification algorithm implementation  Verilog code generated by C2RTL  OpenROAD scripts OpenROAD reports |
| For IPv4 | https://tamimcse.github.io/fib- lookup-linux-kernel | Implementation of SAIL in Linux Kernel  Implementation of SAIL in Domino Routing table data |

# Source code Availability

We have made all our source code and data publicly available on Github. Table 2 shows the links for our projects.

**CPTrie**. We implemented CP-Trie [86], Poptrie [7] and SAIL [184] in C (on Linux). Our source code is publicly available on Github (https://tamimcse.github.io/cp-trie). The source code

4https://gcc.gnu.org/

also contains the routing tables and traffic data. It also contains the performance report generated by our application.

**C2RTL**. We implemented C2RTL [85] as a GCC plugin. The source code of C2RTL is publicly available on Github (https://tamimcse.github.io/c2rtl). The code also contain the C implementa- tion of CP-Trie [86], Poptrie [7] and SAIL [184] based IP lookup and TabTree [117] based packet classification. It also contains the corresponding Verilog code generated by C2RTL. The code also contains OpenROAD [1] scripts to generate chip layout and Performance-Power-Area (PPA). Finally, the link also includes the OpenROAD reports for each of our applications.

**For IPv4**. We used the notion of CP-Trie to improve SAIL [184] based IPv4 lookup in Linux kernel and Domino programmable pipeline based router. The source code is publicly available on Github (https://tamimcse.github.io/cp-trie). It also contains IPv4 routing tables.

# Evaluation metrics

We evaluated routing table lookup algorithms in software and ASIC. In software, measure the following:

* + - We measure the lookup throughput in million lookups per second (Mpls).
    - We measure the memory consumption of the routing table lookup data structure in MB.
    - We measure the update performance in microseconds.

In ASIC, we measure the Performance-Power-Area (PPA):

* + - The performance is measured in clock frequency (GHz).
    - The power is measured in mW.
    - The area is measured in *mm*2

# Organization of the dissertation

The remainder of this dissertation is organized as follows.

* + - Chapter 2 and Chapter 3 describe the background of this dissertation. Chapter 2 describes an overview of a router. It also shows the internal architecture of a router data plane chip. In

this dissertation, we focus on routing table lookup of a router chip. The routing table lookup is performed by a Trie based routing table lookup algorithm. This is why, we also describe an overview of a Trie data structure in Chapter 3.

* + - Chapter 4 and 5 describe the related work on routing lookup algorithms and high-level syn- thesis systems. Chapter 4 discusses the related works on trie based routing table lookup algorithms and implementations. Chapter 5 discusses the related works on high-level synthe- sis tools.
    - Chapter 6 describes CP-Trie based routing table lookup algorithms. Here, we implemented CP-Trie in both software and ASIC
    - Chapter 7 describes C2RTL high-level synthesis tool. Here, we implemented several routing table lookup and packet classification algorithms in software and generated corresponding Verilog code using C2RTL. We then generated chip layout and performance-power-area of the chips using OpenROAD.
    - Chapter 8 describes the use of cumulative PopCount for SAIL based IPv4 lookup. Here, we implemented our algorithms in Linux Kernel and Domino based programmable router.
    - Chapter 9 describes the limitations of this dissertation, and Chapter 10 concludes the disser- tation. Chapter 10 also describes the broader impact and future research of the dissertation.

# CHAPTER 2

**An overview of a router**

The Internet allows billions of devices to communicate to each other. It is not feasible for each device to connect to each other directly. To manage the scalability, the Internet is designed as a network of multiple heterogeneous networks. Figure 2 shows an abstract architecture of the Internet [30]. The Internet consists of two types of networks: access network and backbone network.

* + - **Access Network:** An access network is where end devices are connected. An access network provides connectivity within a region. Examples of access networks include local internet service providers (ISP), enterprise networks, and so on. Access network is also known as edge network. The routers used in the access network are known as edge routers. Example of edge routers include Cisco ASR, Juniper MX and so on.
    - **Backbone Network:** Backbone network is responsible for connecting geographically dis- tributed access networks. Backbone network is also known as core network. Routers used in a backbone network are called core routers. Examples of core routers include Cisco NCS, Juniper PTX and so on.

Edge routers typically require more programmability than core routers. On the other hand, core routers typically require higher throughput than edge router because it needs to carry aggregate traffic. In the past, router vendors used to offer separate platforms for core and edge routers to optimize those objectives. Recently however vendors are offering platforms that offer both high programmability and high throughput. As a result, there is a convergence between edge and core routers. For instance, DriveNet and Arista are developing Broadcom Jericho+ based routers which are used in both core and edge networks [49]. Cisco is also offering Cisco 8000 routers based on SiliconOne chips which can be used as both core and edge routers. Juniper PTX routers based on Juniper Express chips are also used as both core and edge routers.

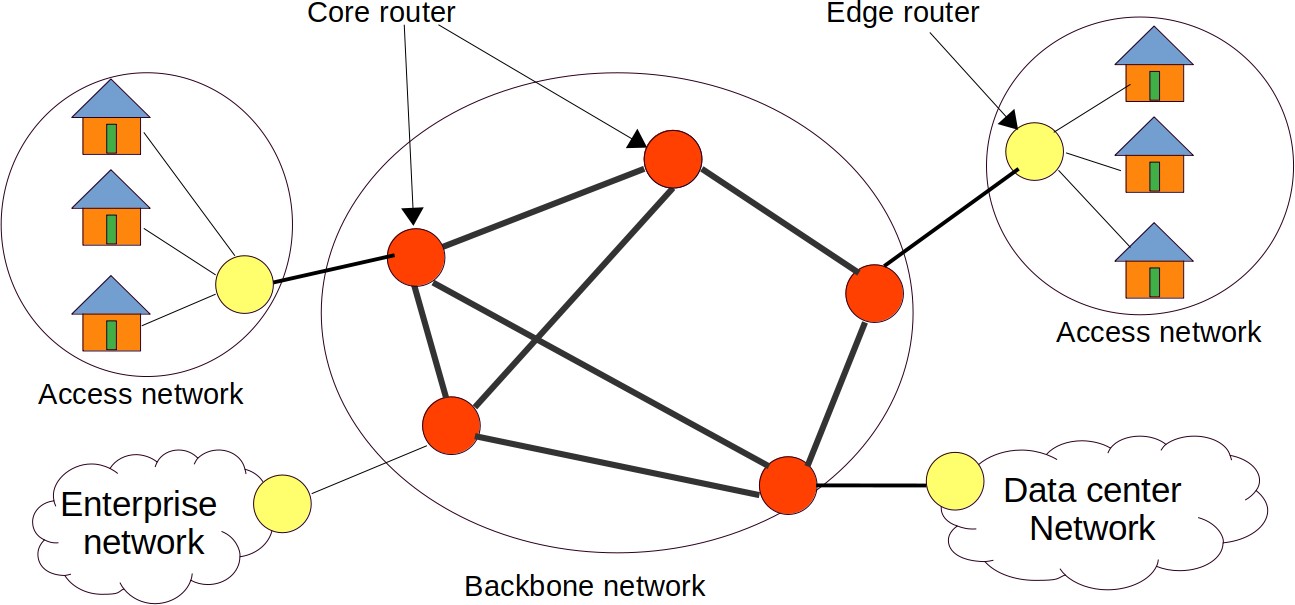


Figure 2: Architecture of the Internet

# Convergence Between Routers and Switches

These days, the terms ”router” and ”switch” are often used interchangeably. This often creates confusions among networking researchers. This section discusses why the terms are often used interchangeably and what are the differences between modern routers and switches.

Routers were primarily designed to connect hosts with different network addresses. Thus, routers primarily support layer-3 forwarding. On the other hand, switches were primarily designed for connecting hosts within a network. Thus, switches primarily support layer-2 forwarding. How- ever, the distinction has become philosophical nowadays [68]. Modern routers can perform both layer-2 and layer-3 forwarding. Again, modern data center and enterprise switches support both layer-2 and layer-3 forwarding. Modern routers and switches however differ in features [37]. For instance, data center and enterprise switches are highly optimized for bandwidth. They consume less power. They typically have very small buffers. They also do not support advanced packet processing. They typically support fewer ports. On the other hand, modern routers support hun- dreds of ports. They support complex packet processing operations. They also offer deep buffers. This is why, they typically offer less throughput than switches. They are also typically less power efficient than switches. Example of switching ASIC include Broadcom Trident (StrataXGS family) [24, 35, 3], Broadcom Tomahawk (StrataXGS family) [22, 23, 177, 164, 35], Barefoot Tofino (now part of Intel) [20, 162], Mellanox Spectrum (now part of Nvidia) [123], Innovium TERALYNX

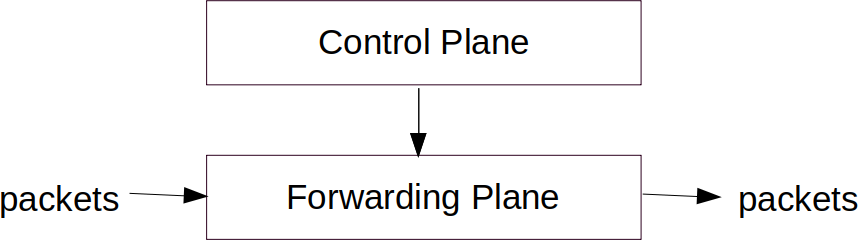


Figure 3: Architecture of a Router

(now part of Marvell), Xsight X1 [182] and so on. On the other hand, examples of routing ASIC include Broadcom Jericho (StrataDNX family) [21, 163, 4, 35], Cisco Silicon One [36, 165], Juniper Express [95] etc. Despite their differences, modern switching ASICs and routing ASICs use very similar architecture [37, 191]. Recently, Cisco is using the same chip architecture for both routing and switching ASIC with Silicon One chip [36]. Alibaba is also developing routers using switching ASIC [167].

# Router Architecture

A router needs to forward several billion packets per second. Implementing a router as a monolithic application does not scale well. A modern router has two types of tasks: 1. control of the router,

1. actual packet processing tasks. Control tasks are not very performance critical whereas packet processing tasks are highly performance critical. This is why, the architecture of a router is divided into two major planes: *control plane* and *forwarding plane* as shown in Figure 3.
   * **Control plane.** The control plane is responsible for router’s control and management func- tionality such as running network protocols (e.g., BGP, IS-IS, OSPF, BFD, IGMP, LDP, etc), setting up filters, tunnels, policers, counters, and so on. The control plane typically runs when the network’s topology or the network’s policy changes. As such events do not occur very frequently, the control plane of a router is generally implemented using software. Such software typically runs a general purpose CPU.
   * **Forwarding plane.** Forwarding plane is responsible for actual packet forwarding. For- warding plane consists of a packet forwarding software and a specialized forwarding chip.

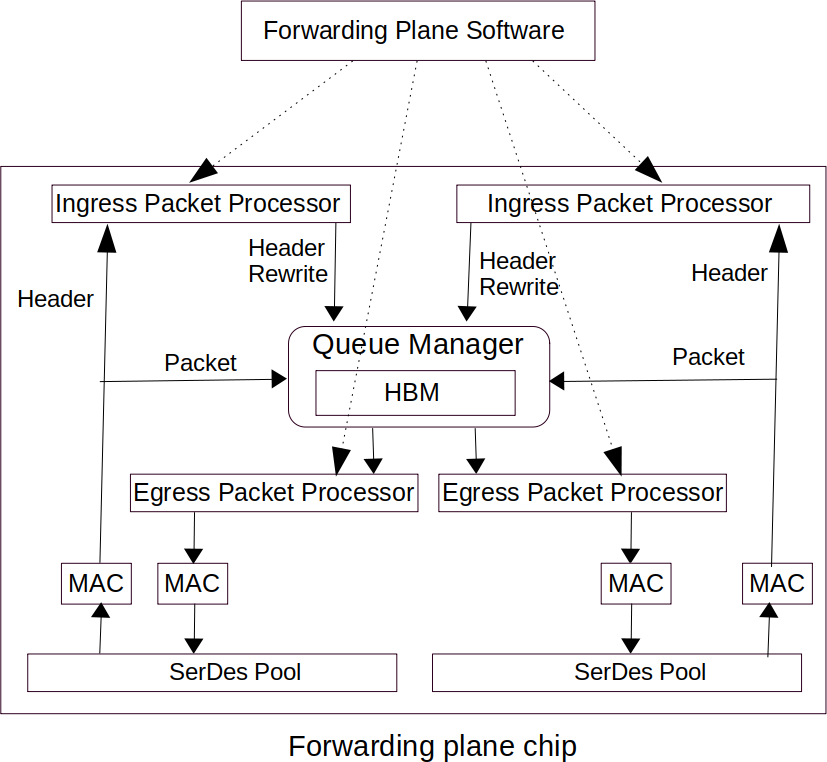


Figure 4: Forwarding plane of a router.

Forwarding plane is sometimes referred to as data plane. Forwarding plane implements ac- tual packet processing logics such as header parsing, source and destination lookup, filter matching, telemetry and so on. A router needs to forward several billion packets per second. Software packet forwarding cannot achieve such throughput. This is why, forwarding plane is generally implemented using specialized ASIC. Note that, an ASIC cannot run itself. Packet forwarding software is also needed to set up the ASIC for packet forwarding. Forwarding plane is the focus of this dissertation.

# An overview of router forwarding plane

Router forwarding plane consists of a forwarding plane software and a forwarding plane chip (as shown in Figure 4). Here, the chip is responsible for the actual packet processing. The forwarding plane software sets up the on-chip memory of the chip in the way the chip expects. The chip simply reads the on-chip memory and executes predefined functions based on the memory content.

So, in order for the chip to work properly, the forwarding plane software plays a very critical role. Figure 4 shows different modules of a chip. A chip can contain multiple pipelines. For instance. in Figure 4, there are two packet processing pipelines. Each packet processing pipeline has its own ingress and egress packet processor, ingress and egress MAC and SerDes pool. The queue manager is shared by all the pipelines. Here, we discuss the packet processing steps:

1: Input signals received at SerDeses are sent to MAC (Media Access Control). MAC turns the bit stream into packets. It finds the start and the end of a frame. It checks Frame Check Sequence (FCS), inter-frame gap, frame preamble to detect any error. It also performs Forward Error Correction (FEC) if needed.

2: The packet is then sent to a Queue manager. The Queue manager contains a High Bandwidth Memory (HBM) which acts as a shared packet buffer. HBM is a special memory that supports very high bandwidth read/write operations. MAC also sends the packet header to the ingress packet processor. Note that the actual packet is not sent to the ingress packet processor.

3: The ingress packet processor is responsible for the actual packet processing. It performs routing table lookup and determines the egress port for the packet. It then notifies the queue manager. It issues a header rewrite request to the queue manager. As the queue manager has the actual packet, it is responsible for updating the header.

4: Queue manager dequeues the packet from the HBM memory. Egress packet processor ac- cepts the packet from the queue manager. Egress packet processor may perform additional operations on the packet before transmitting it to the egress MAC. The egress MAC then transmits the packet to output ports through a SerDes.

Note that, actual packet processing takes place in the ingress and egress packet processor. Here, the ingress packet processor is the most critical block because it performs most of the operations. Egress packet processor performs a few simpler operations. Routing table lookup is performed by ingress packet processor. The ingress packet processor however is not a monolithic block. It consists of several blocks (Figure 5). As the ingress packet processor plays the most critical role inside the data-plane chip, we will look into the details of the ingress packet processor.

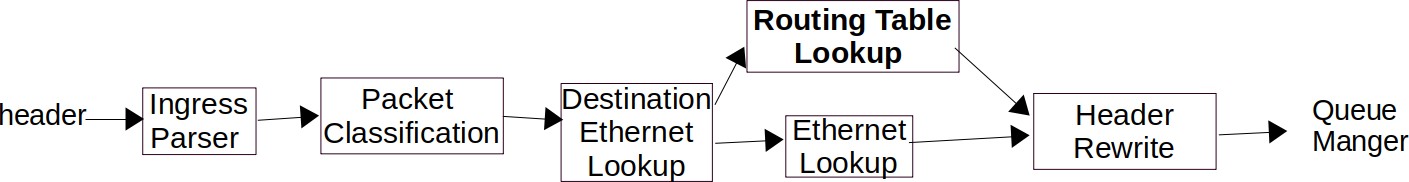


Figure 5: Modules of an ingress packet processor.

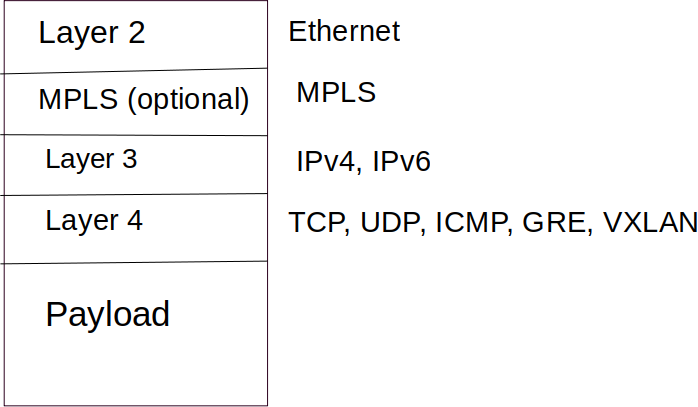


Figure 6: Packet headers.

# Ingress Packet Processor

Figure 5 shows different modules of an example ingress packet processor (similar to [4]). Here, each module needs to process packets at line rate so that ingress processor can process packets at line rate. Here each module acts like a pipeline stage.

1: **Ingress Parser** parses the packet header and generates a header array. The ingress parser knows how to parse IP headers, ethernet headers, layer-4 headers and so on. The header array has a fixed location for each header. Figure 6 shows some example headers. The header array also contains metadata fields such as the input port number, timestamps and so on. The header array is processed by each module. A module may update the header array along the way. Ingress parser sends the header array to **Packet Classification** module.

Table 3: Example ruleset (for packet classification)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Rule | SA | DA | SP | DP | Protocol | Action |
| r1 | 01100∗ | 01100∗ | 111 | 111 | 80 | forward |
| r2 | 11010∗ | ∗ | 10∗ | 11∗ | 22 | drop |
| r3 | ∗ | 01111∗ | 000 | 001 | \* | enqueue |

2: **Packet Classification** module checks if it’s a tunneled packet (e.g. MPLS, GRE, VXLAN, etc). If it is a tunneled packet and is destined to an endpoint on this router, the packet is decapsulated. The decapsulated packet is also parsed. The header array is updated with the new headers. Packet classification may also perform additional access control and policing. Packet classification involves finding the best match for a packet in a multi-field ruleset to determine the action. Table 3 shows an example ruleset of a router. Here, the rules in the ruleset consist of a source address (SA), a destination address (DA), a source port (SP), a destination port (DP) and a protocol field. Each rule in the ruleset also has a priority. Here, we can have exact or prefix match. 5 A packet may match with more than one rule. In that case, the rule with the highest priority determines the action. Note that, a router may be required to execute a chain of packet classification tasks. A packet classification module cannot process such tasks at line rate. This is why, there can be several packet classification modules so that they achieve line rate. A large chain of packet classification tasks may still stall the pipeline. Such pipeline stall results in packet drop in router. Packet classification module sends the header array to **Destination Ethernet Lookup** module.

3: **Destination Ethernet Lookup** module checks if the destination MAC address matches to that of the input interface; if yes, then the packet is a candidate for layer-3 forwarding. It the forwards the packet to **Routing Table Lookup** module. Otherwise, the packet is forwarded to **Ethernet Lookup** module.

5Packet classification rules can also have range match (e,g. 20 *−* 50). However, a range can be represented as one or multiple prefixes [117].

Table 4: Example forwarding table (for IP lookup)

|  |  |
| --- | --- |
| Prefix | Next-hop |
| 169*.*254*.*0*.*0*/*16 | 3 |
| 169*.*254*.*192*.*0*/*18 | 4 |
| 192*.*168*.*122*.*0*/*24 | 5 |

4: **Routing Table Lookup** performs a lookup on the destination IP address. If there is a match, then it knows the port and the egress packet processor for the packet to send to. Routing table lookup is one of the most expensive operations in a router. It is the focus of this dissertation. IP lookup involves performing the longest prefix match of the destination IP address. For instance, Table 4 shows an example forwarding table of a router. If the destination IP address of an incoming packet is 169*.*254*.*198*.*1, then the packet should be forwarded to next-hop 4. Again, if the destination IP address is 169*.*254*.*190*.*5, the packet should be forwarded to next-hop 3.

5: **Ethernet Lookup** module performs lookup on both source and destination ethernet address. Here source ethernet address lookup is used for MAC learning. MAC learning maintains a table that contains the MAC addresses of the neighboring hosts and the port number through which they can be reached. If a source MAC is not present in the table, the table is updated. Ethernet lookup module then performs a lookup on the destination MAC address. If there is a match, then it knows which ports to send to. Otherwise, the packet is forwarded to all ports (bridging).

6: **Header Rewrite** module replaces the L2 headers with the new L2 headers of the packet.

Ingress packet processor is configured by a forwarding plane software (as shown in Figure 4). The forwarding plane software receives change requests from the control plane. It then updates the ASIC memory accordingly. The forwarding plane software does not execute on the ASIC. It run on a separate CPU (usually on the same CPU the control plane software). The main function of packet forwarding software is to configure the ASIC. The internal memories of a dataplane chip

are exposed via registers. Forwarding plane software uses those registers to configure the memory. Sometimes, the forwarding plane software is referred to as an ASIC compiler because they program IP prefixes and packet classification rules into ASIC. They however are not like processor compilers.

# Multiple pipelines in a routing chip

25*.*6∗1012

A 25*.*6 Tb/s router needs to forward around 11*.*6 billion packets per second (Bpps) ( 8 ) for

256+8+12

256B packets (a packet will also require 8 bytes frame header and 12 bytes frame interval) [191].

25*.*6∗1012

64+8+12

Again, for 64B packets, it will need to forward around 38*.*1 billion packets per second ( 8 ).

However, a routing chip’s clock speed is typically around only 1 GHz [191]. A 1 GHz pipeline ASIC can only process 1 billion packets per second (1 packet in every 1 ns). This is why, routing chip typically uses multiple pipelines. Here, multiple pipelines are analogous to CPU cores in a multi-core CPU. Currently, routing chips such as Broadcom StrataDNX, Cisco Silicon One, Juniper Express use multiple pipelines where each pipeline runs at around 1 GHz. Currently, the latest version of Broadcom StrataDNX, Cisco Silicon One and Juniper Express are manufactured at TSMC’s 7nm process technology.

# CHAPTER 3

**An overview of Trie data structure**

Trie is also known as prefix tree or multiway tree. Trie is an efficient information re**Trie**val data structure. Trie is widely used in many real-world applications such as string matching, dictionary, routing table lookup, packet classification, DNA sequence matching, auto correct, auto suggest, lexicographic sorting, and so on.

Using Trie, search complexities can be brought to optimal limit (key length). The lookup complexity of a trie is *O*(1). Although all tries have *O*(1) lookup complexity, not all the tries are the same. In this dissertation, we designed a trie that needs fewer levels than that of the state-of-the-art tries. Thus, it improves the lookup speed by a constant factor. Even though such improvement is not significant in terms of algorithmic complexity, it has some significant real-world benefits. The fewer number of levels results in faster lookup. It also reduces the power consumption in ASIC. The fewer number of levels also results in fewer SRAM blocks needed in ASIC which has significant impact on the performance.

In this dissertation, we described a new trie named CP-Trie for IP lookup. As this dissertation primarily focuses on advanced trie data structures, we begin by discussing a primitive Trie data structure. This chapter shows an overview of a Trie data structure and related terminologies.

Figure 7 shows an example Trie for a dictionary of strings. Here, each child corresponds a character. Any node except the root can be the end of a word. Searching for a string with length *n* in this dictionary can be performed by up to *n* operations. In the same way, prefixes in a routing table also can be represented by a trie. Figure 8a shows an example routing table and the corresponding trie. Note that, trie results in a very compact data structure of a routing table. If an IP address contains *n* bits, then the lookup in trie will require upto *n* operations. Note that, an IPv6 address is 128 bits long. Thus, it will require upto 128 operations for each IP lookup which is not very efficient.

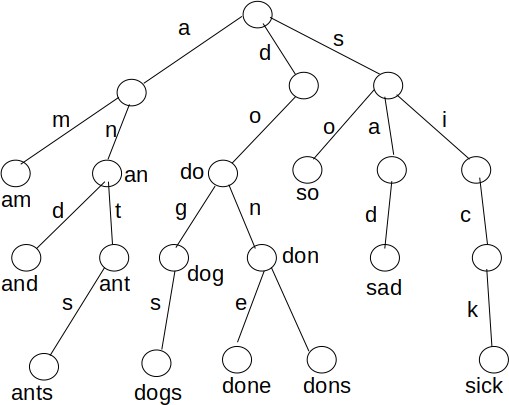


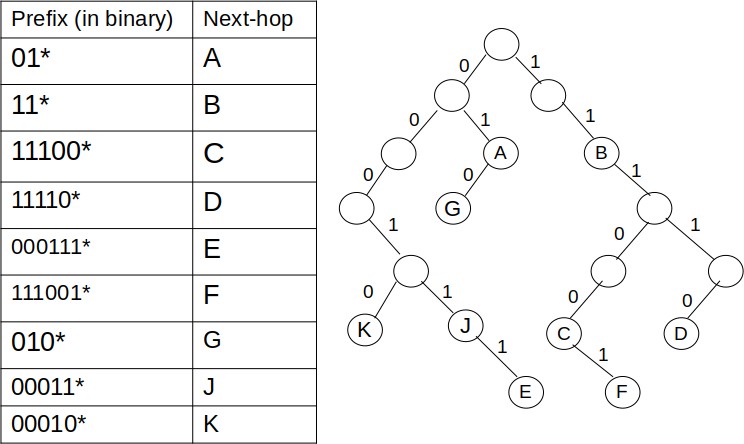
Figure 7: An example Trie for a dictionary consisting of the words ”am”, ”an”, ”and”, ”ant”, ”ants”, ”do”, ”dog”, ”dogs”, ”don”, ”done”, ”dons”, ”so”, ”sad”, ”sick”

**Using longer stride.** In Figure 8a, we use 1 bit stride. However, the depth of the trie can be reduced by using a longer stride. Figure 8b shows the 2-bit stride trie for the routing table in Figure 8a. It shows that using 2-bit stride reduces the height of the trie by half. Thus, it will require half of the operations needed for IP lookup compared to that of Figure 8a. Note that, the height of the trie can be reduced even further by using an even longer stride.

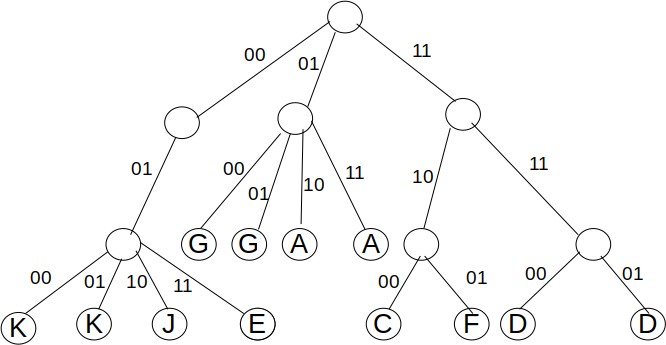
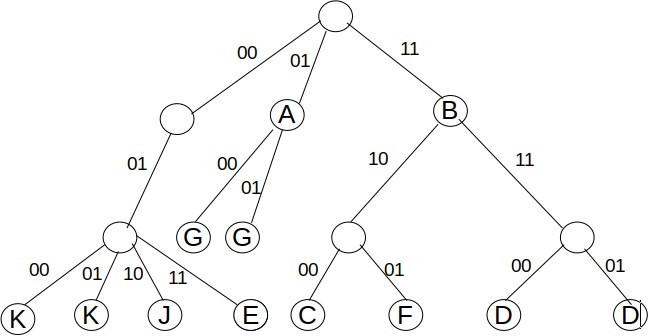
**Leaf pushing.** Note that, in a typical trie, any node except the root can be the end of a prefix. This complicates the lookup process, because lookup needs to keep track of the longest prefix it found so far. This problem can be avoided by leaf pushing [157]. Figure 8c shows the corresponding leaf pushing trie. Here, all the prefixes are pushed to the leaf.

# Encoding Trie with bitmaps

A major limitation of trie is that each node needs to keep track of multiple child pointers. As not all children are present in each node, this consumes lots of unused memory. The nodes in a trie are also allocated dynamically. Thus, they are stored in memory in a disorderly fashion. Such memory allocation cannot be easily mapped to hardware memory. Such disorderly memory allocation is also not not cache friendly on CPU. To overcome the limitations, bitmap based tries have been



* + - 1. A routing table with prefixes and the corresponding trie



* + - 1. Corresponding trie with 2-bit stride (c) Corresponding trie with 2-bit stride with leaf push-

ing

proposed [7, 50]. Here, the presence of leaf children nodes are encoded with bitmaps. Here, the nodes in the same level are also stored consecutively. The lookup algorithm uses the bitmap to calculate the index to the child node. This is how, the trie can be stored in few arrays instead of many nodes. This approach allows us to map the arrays to hardware memories. The arrays are also very cache friendly in CPU. Poptrie [7] proposed a bitmap based trie for IP lookup which can achieve very impressive lookup performance while consuming small memory. Poptrie however can only use upto 6-bit stride because it uses PopCount CPU instruction which can process 64 bit at a time. In this dissertation, we proposed an extension of Poptrie named CP-Trie. CP-Trie stores cumulative PopCount along with bitmaps which allows it to use longer stride (8 − 16-bit stride). Thus, it improves the lookup performance by reducing the number of levels. It will also need fewer

SRAM blocks in ASIC, thus it is a more practical solution for ASIC.

# CHAPTER 4

**Related works on Trie based IP lookup algorithms and implementations**

Trie based solutions are very attractive for solving longest prefix match problem due to small memory consumption and constant lookup performance. Figure 9 shows some of the most significant of Trie based IP lookup algorithms in the past two decades. Trie based IP lookup has been adopted in core routers from Cisco, Juniper and Arista [50, 93, 5].

LC-trie [129] combines path and level compression in a trie to reduce memory consumption. It however exhibits poor cache behavior resulting in slow lookup speed [54]. LC-trie has been adopted in Linux kernel. R´etv´ari et al. [140] proposed a trie compression algorithm that can compress a FIB with 440K routes with just 100-400 KB. However, a small FIB table does not always result in good lookup performance. For instance, their IPv4 FIB lookup consumes 194 CPU cycles. SAIL [184] proposed a trie splitting approach where a routing table is split into multiple levels. It encodes each level with an array. Although SAIL shows impressive lookup performance, it consumes very large memory. Islam et al. [84] observed that most of the array elements of SAIL remain zero in practice. They used bitmaps to keep track of non-empty elements. This approach eliminates all the empty elements which reduces the memory consumption of SAIL by up to 80%. They also implemented their solution inside Linux kernel and Domino [149] switch compiler. Lule˚a [47] encodes a leaf-

2021

2015

2014

2011

2009

2004

1999

1997

CP-Trie [HPSR]

SAIL [SIGCOMM]

SST [ICNP]

LC-Trie [JSAC]

Poptrie [SIGCOMM]

OET [INFOCOM]

Tree Bitmap [SIGCOMM CCR]

Lule˚a [SIGCOMM]

Figure 9: Trie based IP lookup

pushing tree [157] with bitmaps. This scheme reduces the memory consumption significantly. However, the incremental update in this approach can be very expensive because a prefix with smaller length can be pushed to upper levels. Tree Bitmap [50] improves the update performance of Lule˚a by limiting the number of levels a node can be pushed. These scheme however increases the memory consumption moderately compared to Lule˚a algorithm. TreeBitmap has been adopted by Cisco routers. Shape-Shifting Trie [154] is a variant of Tree Bitmap where it encodes a Trie using *shape bitmap* and compact tree [87]. This reduces the memory consumption of Tree Bitmap by approximately 50% for sparse tries. FlashTrie [15] combines hashing and prefix-compressed trie (PC-trie) which reduces the memory consumption significantly. Offset Encoded Trie (OET) [76] encodes each trie node with a next-hop bitmap and an offset value to reduce memory consumption. Helix [141] used parallel search on different levels of a trie to improve IP lookup and update speed. However, their approach requires very large memory. Poptrie [7] has been proposed which reduces the memory consumption significantly while exhibiting impressive lookup performance. The main shortcoming of Poptrie is that it uses 6 bit stride. This dissertation presents CP-Trie where it uses 16-bit stride for the first level and 8-bit stride for the following levels. This reduces the number of steps and the number of memory access needed for a routing table lookup.

# 4.0.1 Trie based IP lookup implementations

Several researches have described specialized hardwares for implementing Trie based IP lookup algorithms. LEAP [70] and PLUG [45] proposed programmable hardwares which are specialized for implementing various IP lookup and packet classification algorithms. SWSL [100] and C2RTL

[85] proposed high-level synthesis tools which can generate ASIC (Verilog RTL) directly from an IP lookup algorithm implemented in C/C++. Several proposals [12, 16, 94, 107, 72, 91, 89] proposed different Trie splitting approaches for ASIC so that all the SRAM modules have uniform amount of memory. This is done in order to avoid bottleneck among the pipeline stages. In this dissertation, we implemented CP-Trie and Poptrie in pipelined ASIC using C2RTL, a high-level synthesis tool. [85].

# CHAPTER 5

**Related works on high-level synthesis**

High-level synthesis is a long standing research topic.

**Early history.** There have been extensive research on on high-level synthesis of digital circuits from late 1970s to early 1990s [39]. Most of the fundamental HLS design steps and fundamental algorithms were designed during that time. These early research projects helped to create a basis for algorithmic synthesis with many innovations which are still being used to this day. However those innovations did not lead to the adoption of HLS during that time. This is primarily because the RTL synthesis was not widely adopted at that time, thus HLS built on top of RTL synthesis, did not have a sound foundation in practice.

**In 1990s.** With improvements in RTL synthesis tools, deployment of HLS tools became more practical. Formal model of digital circuit synthesis led to practical HLS tools [46]. Early HLS tools mainly used behavioral Verilog or VHDL to generate RTL level Verilog or VHDL code. These tools include Mentor’s Monet [51], Synopsys’ Behavioral Compiler [102], IMPACT [98] and so on. Several semiconductor companies including IBM, Motorola, Siemens, etc also developed their in-house high level synthesis tools [39]. However, they did not received widespread adoption because behavioral HDLs were not popular among algorithm designers.

**In the 2000s.** Since 2000, new generation of HLS tools focused on system design based on C, such as, Hardware C [106], Handel-C [151], SystemC [65], SpecC [55], SystemVerilog [79], Bluespec [18] and so on. However, the lack of standard of programming languages slowed down the adoption of HLS [166].

**Recent effort.** The latest generation of HLS tools primarily use C or SystemC for system design. Here, C is more popular among FPGA developers whereas SystemC is more popular among ASIC developers because it enables more hardware specific control. However, both C and SystemC can be used to generate RTL code for both ASIC and FPGA.

Several projects used other high-level languages to write RTL code rather than Verilog and VHDL. For instance, Chisel [13] allows us to develop hardware in Scala. Again, PyMTL [122] allows us to design hardware in Python. SystemVerilog [79] and Bluespec [18] allow us to design hardware in a custom higher level language. Chisel, PyMTL, SystemVerilog and Bluespec are quite popular among hardware designers [131] because they allow RTL developers to be more productive. However, they are not popular among software developers because the code here is written in RTL fashion rather than algorithmic fashion. This is why, these frameworks are not considered high-level synthesis.

Despite such a long history of HLS, HLS has not been widely adopted to these days. Current generation of HLS tools for ASIC include Mentor’s Catapult [41, 53], Cadence’s Stratus [28] (for- merly C-to-Silicon Compiler [103] and Forte’s Cynthesizer [143]), NEC’s CyberWorkBench[175], Synopsys’ Synphony (formally HP PICO [146]) and so on. These tools however are not open source. There is no study on the effectiveness of these tools for IP lookup and packet classification algorithms, to the best of our knowledge. Recently, Google is working on an open source high-level synthesis tool named XLS 6 which is still a work in progress. Alladin [147] proposed an open source accelerator simulator which takes a C function as an input and generates power and area without creating the actual RTL. Alladin does not support control-flow intensive programs, thus can not be used for IP lookup and packet classification algorithms.

There are several high-level synthesis systems for FPGA [126]. These tools include Xilinx Vivado (formerly AutoPilot [39], originated as xPilot [31] at UCLA), Intel HLS compiler [80], Bambu [133], LegUp [29], SPARK [66], GAUT [40], DWARV [125], VAST [188], Altera C2H [38],

ROCCC [174], Kiwi [148], CASH [26], Trident [170], ASC [124], and so on. These solutions however cannot generate RTL for pipelined ASIC, at this moment. More recently, FPGA vendors such as Xilinx and Intel propose to use parallel programming model OpenCL for FPGAs to improve the programmability and productivity. For instance, Xilinx SDAccel and Intel’s SDK For OpenCL can generate RTL code from OpenCL [180, 81].

6https://github.com/google/xls

# 5.0.1 High-level Synthesis for IP Lookup

A routing chip differs from traditional chips in two ways. Firstly, it needs to support pipelining so that it can process packets at line rate. A routing chip may need to process several billion packets per second. Secondly, a routing chip needs to implement control flow intensive applications because many router applications such as IP lookup and packet classification algorithms need to implement many nested conditional statements.

There has been relatively little work on high-level synthesis for IP lookup and packet classifi- cation. SWSL [100] is an academic high-level synthesis tool that takes an IP lookup program as input and generates synthesizable Verilog RTL. SWSL was developed based on the observation that most network lookup algorithms consist of simple algorithmic steps where each step only accesses variables from its own state. SWSL converts each step to a Verilog module. SWSL has been used to implement simple trie based IP lookup. However, SWSL appears to be abandoned at this moment.

Several research projects used high-level languages such as C and C++ to program programmable pipeline based ASIC. LEAP [70] and PLUG [45] proposed hardware accelerators and compilers where network lookup or packet classification algorithm can be implemented in a high-level lan- guage (C/C++) and compiled to the programmable hardware accelerator. LEAP and PLUG have been used to implement IP lookup and packet classification in hardware [70, 45, 173]. Domino [149] presents a switch compiler which takes a high level program written in C as an input and generates machine code for programmable pipelined based switches. Domino has been used to implement IP lookup algorithm in switching ASIC[84]. Recently, Chipmunk [61] used program synthesis to produce switch machine code from a C program. They used their method to generate machine code for Barefoot’s Tofino switch [20]. In this dissertation however, we mainly focus on fixed function ASIC rather than programmable ASIC.

This dissertation presents a high-level synthesis tool named C2RTL. We implemented several IP lookup and packet classification algorithms in C2RTL. Our experiment shows that C2RTL can be instrumental in designing IP lookup and packet classification in pipelined ASIC. We also evaluated the resulting Verilog RTL using OpenROAD. Thus, C2RTL+OpenROAD flow allows us to generate physical chip layout and area/power characteristics of different algorithms implemented in C. This approach can be instrumental in evaluating different algorithms at system level.

# CHAPTER 6

**CP-Trie: Cumulative PopCount based Trie for IPv6 Routing Table Lookup in Software and ASIC**

This dissertation presents an IPv6 routing table lookup algorithm named CP-Trie which is an extension of Poptrie [7]. Poptrie divides an IPv6 routing table table into 20 levels: level 16, 22, 28, 34, 40, 46, 52, 58, 64, 70, 76, 82, 88, 94, 100, 106, 112, 118, 124 and 130 (6-bit stride). 7

It uses 6-bit stride because the PopCount CPU instruction it uses can process only 64 bits on a 64-bit machine (26 = 64). This dissertation presents an extension of Poptrie named CP-Trie where we store cumulative PopCount along with the population counting bitmaps. Using cumulative PopCount enables CP-Trie to use longer stride (8 or 16 bits in this case) while using PopCount instruction only once at each step. CP-Trie splits an IPv6 routing table into 15 levels: level 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120 and 128. As a result, CP-Trie needs fewer number of steps. This results in faster lookup and low power consumption in ASIC. The fewer number of steps also reduce the number of memory accesses. As a result, CP-Trie will require fewer SRAM blocks compared to Poptrie which results in lower area.

This dissertation also describes the implementation of CP-Trie and Poptrie [7] in software and ASIC. We evaluated our software implementation with routing tables from real core router [142]. On the other hand, we evaluated our ASIC implementation (Verilog RTL) using OpenROAD [1] electronic design automation (EDA) tool. We made our software and ASIC implementation publicly available (https://tamimcse.github.io/cp-trie).

A trie based IP lookup algorithm should achieve following design goals:

* **Fewer number of steps**: The fewer number of steps results in faster lookup. It also reduces the power consumption in ASIC.
* **Fewer memory accesses:** Fewer memory accesses is very critical in pipelined ASIC. In

7We use direct indexing for level 16[7]

pipelined ASIC, each pipeline stage has to access SRAM in parallel. Thus, a pipelined ASIC requires separate SRAM blocks which can serve the pipeline stages simultaneously. 8 It also has been found that fewer large SRAM blocks are more area efficient than many smaller SRAM blocks [134, 135].9 This is why, reducing the number of memory access reduces the number of SRAM blocks needed in ASIC which results in a smaller area.

* **Small memory consumption and fast incremental update:** An IP lookup algorithm should consume small memory and fast incremental update.

# Proposed Approach

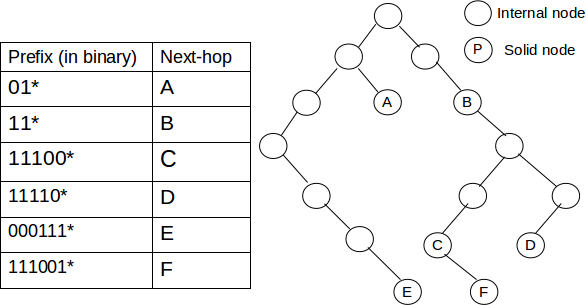
Figure 10 illustrates CP-Trie with a simple example. Figure 10a shows an example routing table and the corresponding binary tree. Here, the nodes with a next-hop are called solid nodes and the rest of the nodes are called internal nodes. For the sake of simplicity, here we split the tree into two levels: level 3 and 6. That is, we process 3 bits in each step (i.e. stride size is 3). Here, we also assume that PopCount instruction can process only 4 bits at a time. That is, the PopCount instruction cannot process the whole stride (23 bits) at a time. CP-Trie pushes the nodes to level 3 and 6 as in Figure 10b. The level pushing is done as following: all the nodes (both solid and internal) in level 1 and 2 are pushed to level 3. If the original tree has an internal node in level 3, those nodes and the nodes in level 4 and 5 are pushed to level 6. For instance, node B is pushed to both level 3 and level 6. CP-Trie then encodes level 3 and 6 with bitmaps as in Figure 10c. Here the *i*-th bit in *B*3 indicates if the *i*-th node in level 3 is a solid node or not. Again, the *i*-th bit in *C*3 indicates if the *i*-th node in level 3 is a leaf or not. Here the leaf nodes are represented by the 0 bits and the non-leaf nodes are represented by the 1 bits. Again, *B*6 is constructed based on the solid nodes in level 6. This is how, we construct *B*3, *C*3 and *B*6 in Figure 10c. As our PopCount can process only 4 bits a time, we group every 4 bits and add a cumulative PopCount in each group (Figure 10c). Cumulative PopCount contains the number of 1 bit prior to that group. This enables us to compute the total number of 1 bits in the whole stride by applying PopCount

8It is also possible to implement pipelined ASIC using a multiport SRAM. However, developing a multiport SRAM

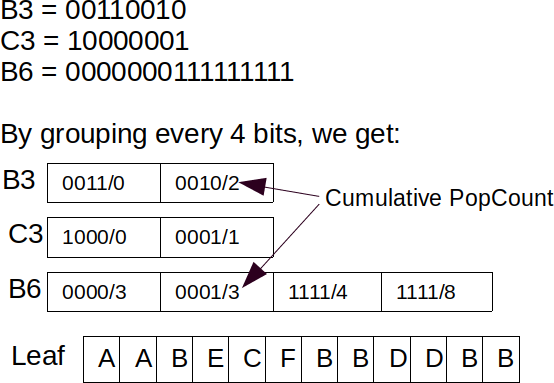
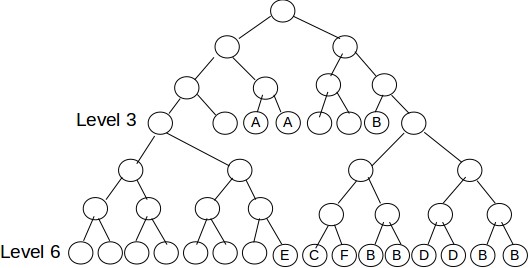
requires designing a custom SRAM cell based on the required number of ports [8, 128]. Increasing the number of ports increases the area and power consumption of the SRAM significantly. This is why, most of the real-world pipelined ASICs use separate SRAM blocks instead of a multiport SRAM [134, 135].

9This is because control logics, address decoders and sense amplifiers of SRAM consume around 85% of the SRAM

area [135].



* + 1. Routing table and the corresponding binary tree



* + 1. Level pushing to level 3 and level 6 (c) Bitmap corresponding to level 3 and 6

Figure 10: CP-Trie data structure

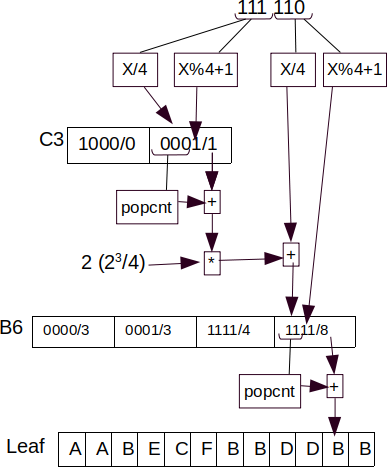


Figure 11: Index calculation for the next level (*B*6) and *Leaf*

only on a part of the stride (this will be discussed more later in this section). Finally, CP-Trie creates a *leaf* array that contains all the solid nodes in level 3 followed by solid nodes in level 6. Note that the cumulative population count of *B*6 does not start from zero. It rather starts from 3 (the total number of 1 bit in level 3). This is because we maintain a single array to store the solid nodes from all the levels. This is why, level 6 needs to start from where level 3 has ended.

**Lookup approach:** Let us assume that we want to lookup an IP address 111110*b* in the routing table. The lookup process is illustrated in Figure 11. The lookup process proceeds as follows. It extracts three leftmost bits from the IP address. Let us call it *x*. Here, *x* = 7. We calculate the

index to *C*3 as *x* . As 7 = 1, *C*3[1] is the corresponding element to the IP address. Note that all

4 4

the division operations in this dissertation are integer divisions. Again we calculate the bit spot inside *C*3[1] as *x*%4 + 1. Here, the bit-spot is 4. That is, the 4-th bit of *C*3[1] corresponds to the IP address. As the bit is 1, it indicates that a longer prefix exists, thus we need to lookup in level

6. We calculate the index to *B*6 as shown in Figure 11. We calculate PopCount of the bits to the left of the spot-bit (the result is 0 in this case) and add that to the cumulative population count (1 in this case). Thus the sum is 1. The number indicates the number of 1s to the left of the spot bit upto *C*3[0]. The number also indicates the number of non-leaf nodes to the left of the current node (in level 3 in Figure 23c). We then multiply the number by 2. This is because each stride is

3

2

3 bits and we group every 4 bits, thus each stride consists of 2 array elements ( 4 ). We then add

it to *x*

4

where *x* is the right-most three bits (6 in this case) from the IP address. Thus the index

to *B*6 is 3 ([*popcnt*(000) + 1] ∗ 2 + 6 ). We also calculate the spot bit inside *B*6[3] as *x*%4 + 1 = 3. Thus, the third bit of *B*6[3] corresponds to the IP address. As the bit is 1, it indicates that the longest prefix match is found. We then calculate the index to *Leaf* as *popcnt*(11) + 8 = 10. Thus the next-hop would be found in *Leaf* [10] (*B* in this case).

4

Note that CP-Trie needs to use just one PopCount instruction at each step even though the stride size is 3. Moreover, if we use longer stride, this approach will still need just one PopCount at each step. In real implementation, we use 16-bit stride for the first level and 8-bit stride for the following levels. We use PopCount instruction that can process only 64 bits. Using cumulative population count saves us from using multiple PopCount instructions for 8 or 16 bit strides. Note that, although it is possible to implement 256-bit PopCount or 65536-bit PopCount in ASIC (for 8 ort 16 bit stride respectively), their area and power consumption would be substantially higher

than that of a 64-bit PopCount [32].

**CP-Trie data structure:** Although we have illustrated CP-Trie with a simpler example, in actual implementation, CP-Trie divides a routing table into 15 levels: level 16, 24, 32, 40, 48, 56,

64, 72, 80, 88, 96, 104, 112, 120 and 128. Thus, each level contains a *B* and *C* array except the level 128 which only contains a *B* array. Each element of *B* and *C* array contains a 64-bit *bitmap* and 32-bit cumulative population count (we refer to as *cumu popcnt* ).

# Lookup Algorithm

Algorithm 1 shows the lookup process of CP-Trie. It starts by extracting 16 leftmost bits from the IP address (Line 3). CP-Trie calculates the index to *C*16 by diving the *stride* by 64 (Line 4). As we group every 64 bits to create a bitmap, the index to both *B* and *C* array is calculated by dividing by 64 (in the similar fashion as in the example where we calculate it as *x* ). Line 5 calculates the bit spot inside the bitmap by taking modulo operation by 64 (whereas in the example, we calculate as *x* % 4 + 1). Line 6 initializes the current level to level 16. Line 8 checks if the matching node in the current level is a non-leaf; if yes, it visits the next level in Line 9 − 13 (will be discussed later). On the other hand, if the matching node in the current level is a leaf, Line 8 will exit the loop. Line 14 then checks if there is a matching prefix for the IP in *B* array of the current level, if yes, Line 15 calculates the index to *Leaf* array in the same fashion as shown in Figure 11. Line 16 returns the next-hop. On the other hand, if Line 14 does not find a matching prefix, it indicates that there is no matching prefix for the IP address. It returns the default next-hop in that case (Line 17).

4

Line 9 − 13 illustrates how CP-Trie visits each level in order. It first extracts the next 8 bits from the IP (Line 9). Line 10 calculates the index to the next level in the same fashion as shown in Figure 11. One difference however is we multiply the sum of cumulative population count and *popcnt lft*() by 4. This is because, as the stride size is 8, each stride consists of 4 array elements

8

2

( 64 ). Line 11 calculates the spot bit based on the current stride. Line 13 updates the current level

to the next level. Note that, the next level of level 16 is level 24. Again, the next level of level 24 is level 32. This is how, we visit level 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120 and 128 in order. Note that CP-Trie does not need to backtrack to a lower level in order to find the longest match. For instance, if the longest prefix match for an IP address is in level 48, CP-Trie will visit *C*16, *C*24, *C*32, *C*40, *C*48 and *B*48.

**Algorithm 1** CP-Trie based routing table lookup algorithm. The function *extract*(*ip, off, len*) extracts a bit array of length *len* from the IP address *ip*, starting from the offset *off* . The function *popcnt lft*(*x, n*) counts the number of 1-bit in left-most *n* bits of bitmap *x*.

**Input**: Destination IP address of the packet - *ip*

**Output**: Next-hop - *nexthop*

1: **procedure** Lookup(*ip*)

2: *mask* ← 0*X*8000000000000000*ULL*

3: *stride* ← *extract*(*ip,* 0*,* 16)

4: *idx* ← *stride /* 64

5: *bitspot* ← *stride* % 64 6: *curr level* ← *level*16 7: *offset* ← 16

8: **while** *curr level.C*[*idx*]*.bitmap* & (*mask >> bitspot*) **do**

9: *stride* ← *extract*(*ip, offset,* 8)

10: *idx* ← (*curr level.C*[*idx*]*.cumu popcnt* + *popcnt lft*(*curr level.C*[*idx*]*.bitmap, bitspot*)) ∗

4 + *stride /* 64

11: *bitspot* ← *stride* % 64

12: *offset* ← *offset* + 8

13: *curr level* ← *curr level.next*

14: **if** *curr level.B*[*idx*]*.bitmap* & (*mask >> bitspot*) **then**

15: *idx* ← *curr level.B*[*idx*]*.cumu popcnt* + *popcnt lft*(*curr level.B*[*idx*]*.bitmap, bitspot*)

16: return *Leaf* [*idx*]

17: return *default nexthop d* //return default next-hop if no match is found

# Implementation

This section discusses the software and ASIC implementation of CP-Trie and Poptrie. Our soft- ware and ASIC implementation will be found in Github (https://tamimcse.github.io/cp-trie). This experiment has been conducted on a machine that has an Intel(R) Core (TM) i7-4510U processor and 16 GB DRAM. The processor has 2 cores with 4 threads running at up to 3*.*1 GHz (maximum frequency). The processor has 32 KB L1 data cache, 32 KB L1 instruction cache, 256 KB L2 cache and 4 MB L-3 cache.

# Software Implementation

We implemented Poptrie, CP-Trie and SAIL (both SAIL U and SAIL L)[184] based FIB lookup and FIB update. Our implementation contains around 3*,* 000 lines of C code. Here, we used the results of SAIL U to perform unit testing of our implementation of Poptrie and CP-Trie. We choose SAIL U as the basis of our testing because of its simpler implementation. However, as the performance of SAIL is significantly different from that of Poptrie and CP-Trie, we do not compare SAIL in this dissertation. In this dissertation, we use CPU performance counter (TSC register) to measure the performance. 10

# ASIC Implementation

We used C2RTL [85] high-level synthesis (HLS) tool to generate synthesizable Verilog implemen- tations of CP-Trie and Poptrie. C2RTL takes a C implementation of an algorithm as an input and generates corresponding Verilog RTL code. We performed the functional verification of the generated Verilog code of Poptrie and CP-Trie using Icarus Verilog11.

10https://en.wikipedia.org/wiki/Time Stamp Counter

11<http://iverilog.icarus.com/>

Table 5: Power, area and timing in ASIC (for logic)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Poptrie | CP-Trie | Improvement (%) |
| Clock speed | 1 GHz | 1 GHz | Same |
| Internal Power | 76*.*5 mW | 64*.*6 mW | 15% |
| Switching Power | 24*.*4 mW | 22*.*2 mW | 9% |
| Leakage Power | 1*.*15 mW | 0*.*926 mW | 14% |
| Total Power | 102*.*05 mW | 87*.*726 mW | 14% |
| Area | 0*.*0658 *mm*2 | 0*.*0523 *mm*2 | 20.5% |

# Evaluation of ASIC

An algorithm in ASIC is evaluated based on the area and power. EDA tools such as OpenROAD can generate the power and area directly from the Verilog code. Here we used OpenROAD to evaluate the area and power of ASIC.

EDA tools enable us to generate physical chip layout directly from Verilog RTL code. They also provide us with the area and power. Examples of EDA tools include OpenROAD [1], Synopsis Fusion compiler [161], Cadence Innovus [27], and so on. They take Verilog RTL code, macros (e.g. SRAM, I/O ports, etc), a standard cell library [158] 12 and constraints (e.g. clock cycle) as inputs and generates corresponding physical chip layout in GDSII format. Here, we use OpenROAD [1] to generate the physical chip layout of Poptrie and CP-Trie. We pass the following inputs to OpenROAD: 1) an RTL implementation of a FIB lookup algorithm, 2) 1 GHz clock (clock cycle is 1 ns) and 3) CMOS 45 nm Nandgate Open Cell Library. OpenROAD generates GDSII layout from the inputs. Our OpenROAD timing report shows that our implementation of CP-Trie and Poptrie can achieve 1 GHz without timing violation. That is, our implementation of CP-Trie and Poptrie can perform one lookup in every 1 ns. Thus, it can perform 1 billion lookups per second. Note that, currently there is no SRAM available for 45 nm Nandgate cell library, to the best of our

12Standard cell is an ASIC development methodology where all the logic operations are converted into standard cells. Thus, designers do not need to worry about low level implementation of the logic. Designers implement their logic in a high level abstraction. The high level logic is translated into low-level standard cell using logic synthesis tools.

knowledge. This is why, we use registers to implement the arrays. OpenROAD also reports power and area of the ASIC. Table 9 shows the summery of OpenROAD report for Poptrie and CP-Trie. It shows that CP-Trie consumes less power and less area compared to Poptrie. This is because CP-Trie needs to implement fewer levels than Poptrie. Thus, it will require fewer logic gates in the ASIC.

Our implementation also shows that Poptrie and CP-Trie need 79 and 59 arrays respectively. Thus, Poptrie and CP-Trie need 79 and 59 SRAM blocks respectively. Note that, fewer large SRAM blocks are more area efficient than more smaller SRAM blocks because control logics, address decoders and sense amplifiers of SRAM consume around 85% area [134, 135]. This is why, although CP-Trie consumes 1*.*36 − 1*.*47× memory compared to Poptrie (shown in Section 6.5), we expect

SRAM blocks of CP-Trie to consume less area and power consumption than those of Poptrie. Thus,

we expect CP-Trie to be even more area and power efficient than reported in Table 9 when SRAM is incorporated.

# EXPERIMENTAL RESULTS

This section evaluates the performance of Poptrie and CP-Trie on a general purpose CPU.

Table 6: FIB dataset

|  |  |  |
| --- | --- | --- |
| Name | # of prefixes | Longest prefix length |
| fib0 | 105*,* 363 | 48 |
| fib1 | 102*,* 126 | 48 |
| fib2 | 79*,* 431 | 64 |
| fib3 | 103*,* 067 | 128 |
| fib4 | 105*,* 957 | 128 |
| fib5 | 102*,* 739 | 64 |
| fib6 | 104*,* 235 | 48 |
| fib7 | 100*,* 899 | 64 |
| fib8 | 102*,* 731 | 128 |

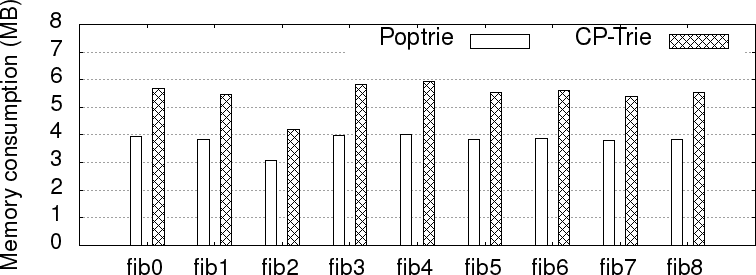


Figure 12: Memory consumption for different FIBs

# Data Set

We obtained the Routing Information Base (RIB) of real core routers from Route Views project [142]. The RIB snapshot was taken on January 17, 2021 at 3 PM EST. We convert the RIB into multiple FIBs based on the autonomous system (AS) IDs. We also keep only one next-hop per prefix in the FIBs. Table 13 shows the details of the FIBs in our data set. Note that, although we performed the experiment on a specific RIB, we found similar results for other RIBs as well. Also note that, edge routers generally have very small routing tables compared to core routers [184]. This is why, we do not use routing tables of edge routers in this experiment.

It is possible to reduce the size of a FIB further by route aggregation [172]. SMALTA [172] and FIFA [121] proposed route aggregation scheme where prefixes belonging to different subtrees are aggregated without modifying forwarding behavior. However, route aggregation does not affect routing table lookup algorithms. Route aggregation can be applied on a FIB without changing the lookup algorithm. Here we do not use *route aggregation* in this experiment.

# Memory Consumption

Figure 12 shows the memory consumption of Poptrie and CP-Trie. It shows that CP-Trie consumes more memory than Poptrie. This is because CP-Trie stores cumulative PopCount along with population counting bitmap. Again, as CP-Trie uses 8-bit stride, it may push a node further than Poptrie. However, this is not always the case. For instance, a prefix with length 48 will be pushed to level 54 in Poptrie, however that node will remain in level 48 in CP-Trie. This is how, CP-Trie may need fewer level pushing than Poptrie for many prefixes. Our experimental results show that

CP-Trie consumes around 1*.*36 − 1*.*47× more memory than Poptrie. The memory complexity of CP-Trie is *O*(*n*) where *n* is the number of IP prefixes in the routing table.

**Choosing the right stride size:** CP-Trie does not impose restrictions on the stride size. One can use any stride higher than 6 in CP-Trie (Poptrie is more suitable if we use stride size 1 − 6). Increasing the stride size reduces the number of levels linearly while increasing the memory

consumption exponentially. Let us assume that the stride size of CP-Trie is *s* where *s >* 6. Then,

each prefix may add 2*s* + 2*s* × 16 bit (each prefix will need 16 bit cumulative PopCount). On the other hand, the number of levels needed is 128 . Ideally, we would like to minimize both 2*s* + 2*s* × 16

64

*s*

64

and 128 where *s >* 6. This is however not possible. Increasing one will decrease the other. This is why, there is a trade-off between the number of levels and memory consumption. Here, we chose *s* = 8, because each prefix may need 40 byte while needing 15 levels. If we chose *s* = 10, each prefix may need 160 byte while needing 12 levels. Note that the stride size is not specific to CP-Trie. One can choose any stride side in CP-Trie. One may also use variable stride sizes, for instance, level 16, 24, 32, 40, 48, 64, 80, 86, 112, 128. Such stride will be beneficial if we have prior knowledge that there will be very few prefixes with lengths 48 − 128. Here, we do not make such assumptions.

*s*

# Traffics for FIB Lookup

We use four traffic patterns in this experiment: prefix, repeated, random and sequential traffic. IP lookup algorithms are typically evaluated based on these traffic patterns [186, 7, 184].

**Prefix traffic:** Our prefix traffic consists of the prefixes in our FIBs. Prefix traffic ensures that the lookups are spread across a FIB.

**Repeated traffic:** Our repeated traffic consists of 224 randomly selected prefixes from the FIBs (with possible duplicates) and adding random bits at the ”don’t care” fields of the prefixes. Each IP in repeated traffic is looked up 10 times repeatedly. Repeated traffic is analogous to a router handling a small number of flows. Repeated traffic is prevalent in real world Internet trace [178]. As we produce the IPs within the range of the prefixes in the FIBs, our repeated traffic is analogous to the real Internet traces. In this dissertation, we do not use actual Internet trace [178]. This is because the real Internet traces were not taken in the same backbone router as the FIBs. Thus, they are not within the ranges of prefixes in FIB.

**Sequential traffic:** Our sequential traffic consists of 28 sequentially increasing IPv6 addresses.

The start address of the sequential traffic is selected such that all the IPs in sequential traffic require matching up to the last level of the FIB. We use sequential traffic to measure the worst lookup throughput for a FIB.

**Random traffic:** Our random traffic consists of 224 random IPv6 addresses within 2000::/4. All the prefixes in our routing table data set also fall in this range. This is why, we choose the range. Note that, random traffic contains IP addresses that are not within the range of the prefixes. Thus, they may not find a match. This is why, random traffic is not a realistic traffic pattern.

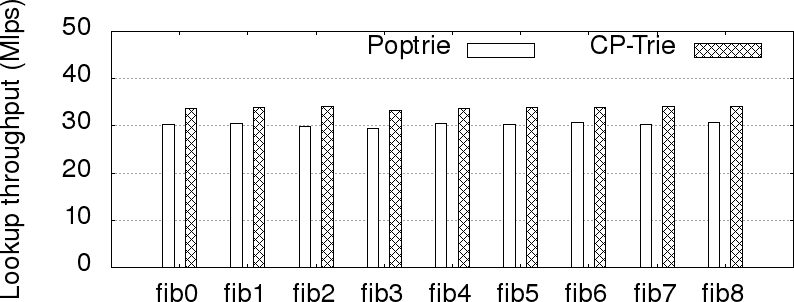
# Lookup Performance in Software

Figure 13 shows the lookup performance in different FIBs for different traffics. We load each traffic in an array in advance and issue a lookup query one by one in sequence. We measure the lookup performance in million lookups per second (Mlps). All the lookup throughputs in this dissertation are measured on a single CPU core.

Figure 13a and 13b show that CP-Trie achieves higher throughput (by around +3 Mpls) than Poptrie for prefix and repeated traffic. This is because, CP-Trie needs to visit fewer levels than Poptrie. Figure 13c shows that CP-Trie outperforms Poptrie by upto 1*.*43× for sequential traffic.

Note that, the lookup throughputs for sequential traffic vary significantly for different FIBs. The

reason for this is as following. We selected sequential traffic such that the matches are found in the last level. As the longest prefix in *fib*3, *fib*4, *fib*8 are 128 bit long (Table 13), both Poptrie and CP-Trie needs to lookup up to the very last level which results in low throughput. On the other hand, the longest prefix in *fib*0, *fib*1, *fib*6 are 48 bit long (Table 13), thus CP-Trie and Poptrie need to lookup up to level 48 and 54 respectively which result in higher throughput than the other FIBs. Again, the longest prefixes in *fib*2, *fib*5, *fib*7 are 64 bit long (Table 13), thus they results in higher throughput that *fib*3, *fib*4, *fib*8, but lower throughput than *fib*0, *fib*1, *fib*6. Thus, 13c shows how lookup throughput can vary based on the length of the prefixes in the FIB. Finally, Figure 13d shows that both CP-Trie and Poptrie achieve significantly higher throughput for random traffic. As random traffic contains many IP addresses that are not within the ranges of the prefixes (hence, not realistic), they only require looking up very few levels in the trie. As Poptrie uses direct match for the first 16 bits, it can detect the unmatch very quickly. CP-Trie does not use such direct matches which results in low throughput compared to Poptrie. In conclusion,



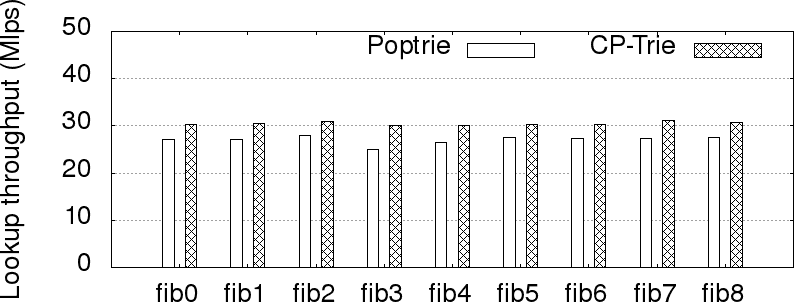
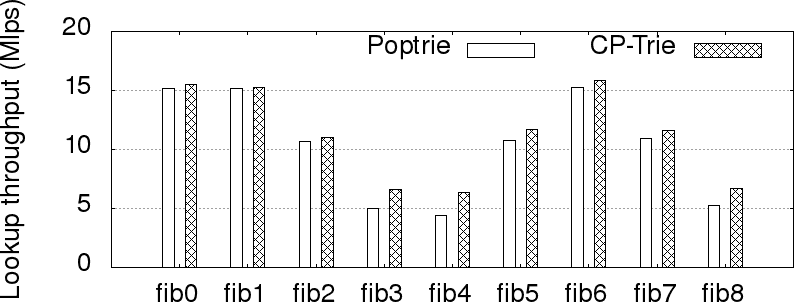
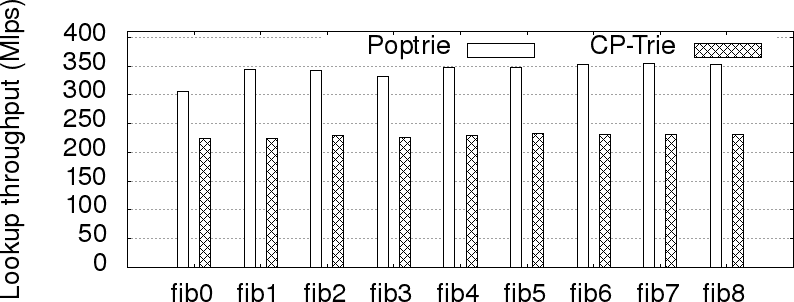
* + - 1. Prefix traffic
      2. Repeated traffic (analogous to real Internet trace)
      3. Sequential traffic
      4. Random traffic (not a realistic traffic)

Figure 13: Lookup throughputs in different FIBs for different traffics

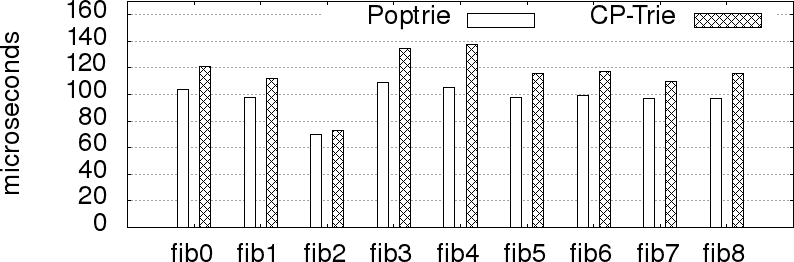


Figure 14: Average insertion time per prefix (smaller is better)

the experiment shows that CP-Trie achieves very high throughput for realistic traffic patterns (such as prefix, repeated and sequential traffic). The time complexity of CP-Trie and Poptrie based IP lookup is *O*(1).

# Update Performance in Software

In order to evaluate the update performance, we measure the insertion time of the all the routes in a routing table. Figure 14 shows the average insertion time per prefix for each routing table. The figure shows that CP-Trie requires slightly more time compared to Poptrie for insertion. The results also show that the update performance of CP-Trie and Poptrie are practically acceptable. The time complexity of CP-Trie and Poptrie update is *O*(*n*) where *n* is the number of IP prefixes in the routing table.

# Discussion

CP-Trie improves the routing table lookup by using a longer stride than Poptrie. CP-Trie can also be useful for Named Data Networking (NDN) lookup. NDN uses the longest prefix match on the names [187, 156]. The names in NDN can be significantly longer than an IPv6 prefix. As CP-Trie uses longer stride, it would be a more practical solution for NDN prefix match compared to Poptrie. However, NDN prefixes are also variable length. Thus, CP-Trie needs to be modified for NDN lookup. Research should be done on the NDN prefix match based on CP-Trie.

# Conclusion

This dissertation presents a cumulative PopCount based routing table lookup algorithm named CP- Trie. CP-Trie is an extension of Poptrie. CP-Trie requires fewer steps than Poptrie which results in higher lookup performance. It also improves the power consumption in ASIC by reducing the number of steps and requiring fewer SRAM blocks. We implemented CP-Trie and Poptrie in soft- ware and pipelined ASIC. Our experimental results show that CP-Trie achieves higher throughput in software and consumes less area and power in ASIC.

# CHAPTER 7

**C2RTL: A High-level Synthesis System for IP Lookup**

Traditionally, ASIC based hardware blocks are implemented in a RTL level HDL such as Verilog or VHDL. However, designing hardware at RTL level is a very complex and tedious process. Here, designers need to use low level abstraction to implement the logic. The RTL code also needs to be cycle accurate. Thus, designers need to consider the path latency of the circuit and insert registers manually to implement pipelining. Such a complex and tedious process calls for designing hardware at a higher level. High-level synthesis (HLS) [41] allows us to design a pipelined ASIC in a high-level language such as C or SystemC [65]. HLS tools generate RTL level hardware logic directly from programs implemented in high level languages such as C or SystemC. This has major advantages over traditional RTL design, such as, increased designer productivity, better complexity management, shorter simulation cycle (no hardware/software codesign is needed), rapid design space exploration, higher quality of results (QoRs) and so on [31, 166, 97]. Despite the benefits, HLS has not been adopted in switching and routing chips. Currently, routing ASICs (e.g. Broadcom Jericho2 [21], Cisco Silicon One [36], etc) are designed in a RTL-level HDL, to the best of our knowledge [33]. Industry typically uses pre-developed intellectual property (IP) instead of HLS because the IPs are well tested [166]. As developing a new IP for a new algorithm requires huge efforts (both in software and ASIC) and any error have very significant financial consequences (there are precedents of such ASIC bugs), industry is often very reluctant to adopt new algorithms in ASIC. As a result, many industrial ASIC is still using decade-old algorithms which have subpar performance.

This dissertation presents a high level synthesis system named C2RTL. C2RTL takes an IP lookup or packet classification algorithm (implemented in C) as an input and generates corre- sponding synthesizable Verilog RTL code. We implemented C2RTL as a GCC plugin. The source code will be found in Github (https://tamimcse.github.io/c2rtl).

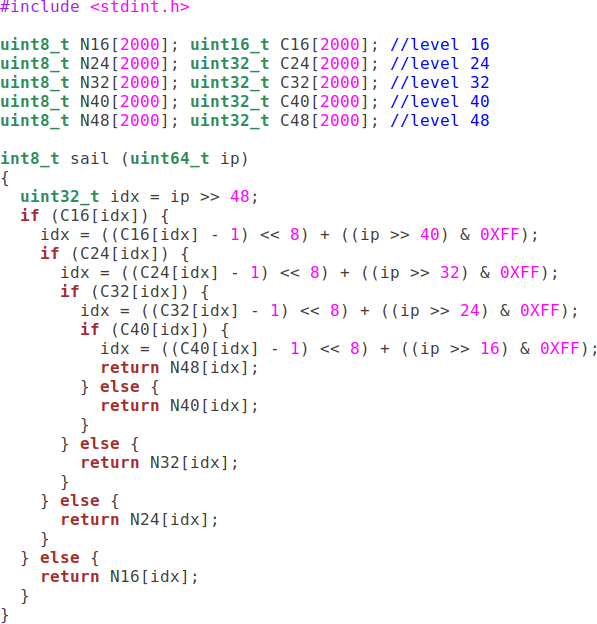


Figure 15: An example trie based IP lookup in C

# IP lookup and Packet classification Algorithms in ASIC

Trie based algorithms have widely been deployed for implementing IP lookup [184, 50, 86, 12, 16,

94, 107, 72] and packet classification [117, 75, 2, 90, 115, 116, 88, 183] in ASIC and FPGA in the past. This dissertation focuses on trie based IP lookup and packet classification algorithms. Figure 15 shows an example trie based IP lookup algorithm named SAIL [184]. Here, the trie is split

among 5 levels. The algorithm first checks level 16. If the node in level 16 has a next-level node, then it will lookup in the next level (level 24). If the node in level 24 has a next-level node, then it will lookup further in the next level (level 32). Thus, the lookup algorithm traverses level 16, 24, 32, 40 and 48 in order until it finds a leaf (the details would be found in Yang et. al. [184]). This is why, the IP lookup algorithm requires several nested conditional statements (*if* statements within *if* statements). Note that, here we only showed lookup upto level 48. An actual IPv6 lookup algorithm needs to visit up to level 128 as an IPv6 address is 128 bit long. Thus, an actual IPv6 lookup algorithm will need many more nested conditional statements. This is why, trie-based IP lookup algorithms can be characterized as a control-flow intensive (CFI) application. Trie based packet classification also falls into the same category. They also do not require a loop. IP lookup or packet classification algorithm also needs to be implemented as a pipeline so that resulting hardware can process packets at line-rate.

Currently, open source HLS tools such as Bambu [133] and LegUP [133] do not support pipelin- ing. This is why, although they support control-flow intensive applications, they cannot be used for IP lookup and packet classification algorithms. This motivated us to design C2RTL. C2RTL is designed to generate pipelined ASIC for control-flow intensive applications.

Table 7: programming restrictions in C2RTL

|  |
| --- |
| No loop (while, for, do-while) |
| No unstructured control flow (goto, break, continue) |
| No ternary operation |
| No dynamic memory allocation |
| No global variables |
| No structure |
| No switch |
| No function call |
| Each branch has to have a separate return statement |

# Programming Convention

An input program in C2RTL is implemented in C language, but with several restrictions (Table 7). Although C allows us to implement an algorithm in different ways, C2RTL requires us to adhere to a programming convention. These restrictions allow us to produce intermediate code (via GCC compiler) in a specific format. Note that, other HLS tools often impose similar restrictions [53, 133]. These restrictions generally come from the underlying compiler. Some of the restrictions also come from the hardware architecture.

C2RTL takes a C function as an input. For instance, Figure 17a shows an example C function which will be processed by C2RTL. Note that, all the arrays here are passed as function argu- ments. The arrays can be implemented as a SRAM block or mutiport SRAM or mutiport register file. Similar coding convention has also been adopted in many other HLS tools including Mentor Catapult, Bambu and so on [53, 133]. As we pass all the inputs and arrays as function parameters, we do not support global variables. C2RTL also requires each branch to have a separate *return* statement (as shown in Figure 17a). We found that GCC does not produce *φ* operation [160, 159] properly if we do not have a separate *return* statement in each branch. This is why, although C allows us to merge multiple branch statements (i.e. having a single *return* for multiple branches), this is not not allowed in C2RTL. Note that, other HLS systems also impose similar restrictions. For instance, Mentor Catapult requires multiple branch statements to have just one single *return* statement (opposite of what C2RTL expects) [53]. C2RTL also does not support *loops*. A *loop- condition* that can only be evaluated at run-time cannot be implemented as a pipeline ASIC. This is why, we do not support *loops* in C2RTL. Again, recursion, dynamic memory, etc cannot be used in hardware, thus are not supported in C2RTL.

# System Overview

There is a formal method of High-level synthesis. In this formal method, the input specification and the circuit are represented as a control and data flow graph (CDFG) [46, 41, 51, 53]. High-level synthesis tools use the formal method to generate RTL code from input specification. A HLS tool needs to perform following tasks:

1. Generate the CDFG from the input.
2. Allocate a resource to each node (operation) of CDFG

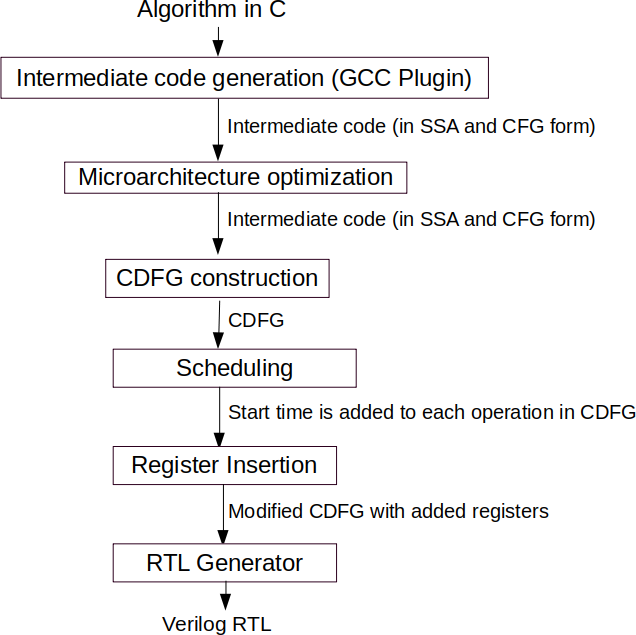
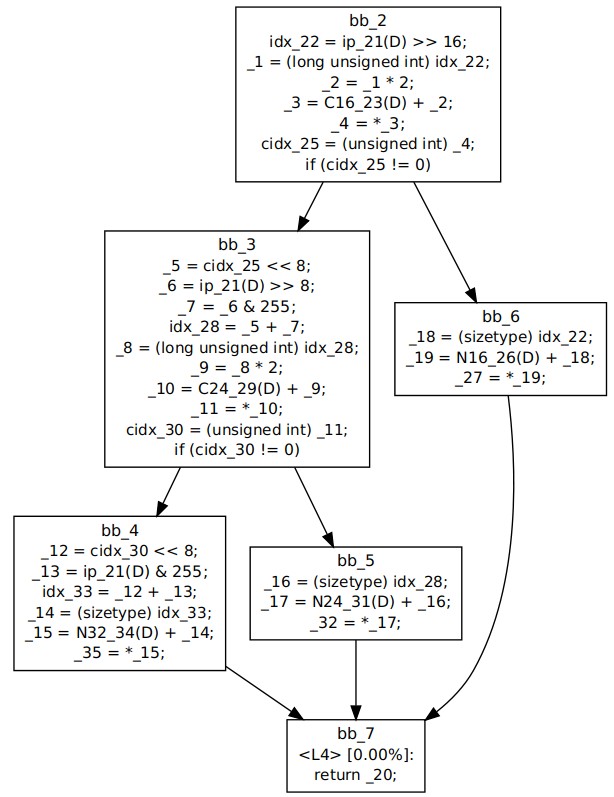
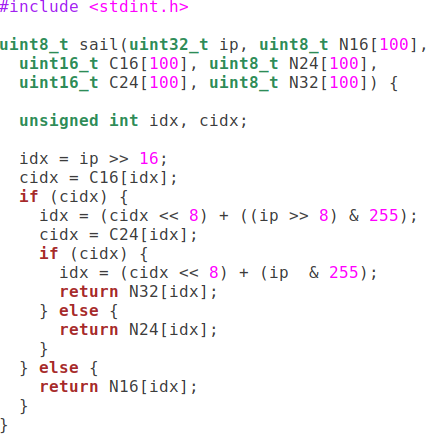


Figure 16: C2RTL design flow

1. Schedule the nodes based on the latency of the resource
2. Binds the operations to functional units, variables to storage elements, and transfers to wires.
3. Generate the RTL instruction set architecture.

In this dissertation, we implemented the formal method of High-level synthesis. Figure 16 illustrates the design flow of C2RTL. Here, the design flow is implementing the above tasks.

**Step 1: Intermediate code generation.** C2RTL leverages GCC compiler to produce inter- mediate code from a C function. GCC takes a C function as an input and generates intermediate code in the form of a control flow graph (CFG). Figure 17b shows the CFG constructed from the example code shown in Figure 17a. Each node in CFG is a basic block. A basic block contains



* 1. Example IPv4 Lookup in C (b) Intermediate code in SSA and CFG form

Figure 17: Intermediate code generation by GCC

intermediate code that can be executed sequentially without any branching. For each if-else block in Figure 17a, two new basic blocks are created in Figure 17b. Edges between basic blocks represent the transfer of control flow from one basic block to another. Here, each basic block contains inter- mediate code in Static Single Assignment (SSA) [160] form. SSA assigns each variable a different name with a version field. When a variable is assigned to a new value, the version field is increased. Thus, SSA guarantees that no versioned variable is assigned twice. This is how, it removes the Write-After-Read and Write-After-Write dependencies. There are only Read-After-Write depen- dencies remain in SSA [60]. Again, as our input program has multiple control flows, SSA adds a *φ* function [160] in the last basic block of the CFG (shown as *< L*4 *>* in *bb 7* ). The *φ* function selects one of the input flows to produce the output (the other flows will not execute). C2RTL obtains the intermediate code along with CFG from GCC using plugin API [62].

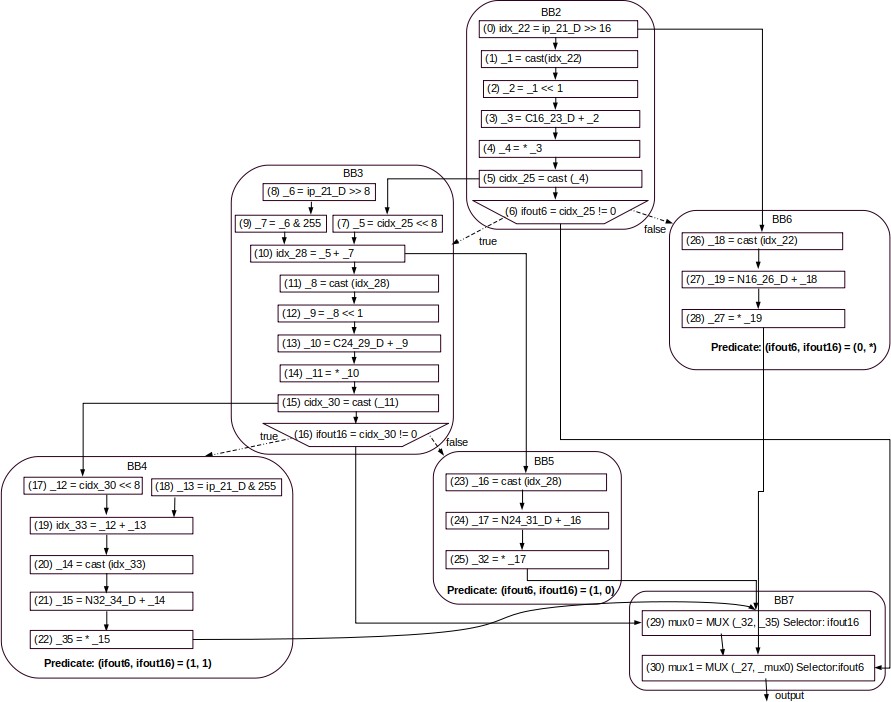


Figure 18: CDFG corresponding to Figure 17b

**Step 2: Microarchitecture optimization.** C2RTL performs standard compiler optimiza- tions such as replacing multiplications and divisions with bitwise operations, operand width reduc- tion, copy propagation and so on.

**Step 3: CDFG construction.** Control and data flow graph (CDFG) is the de facto standard of circuit representation in many HLS tools [98, 111, 53, 147, 175, 103]. C2RTL constructs a CDFG from the intermediate code. Figure 18 shows the CDFG constructed from the intermediate code shown in Figure 17b. There are two types of vertices in CDFG: basic block and operation vertex. A basic block vertex represents a basic block (shown as a rounded rectangle in Figure 18). A basic block consists of a list of statements that can be executed without branching. An operation

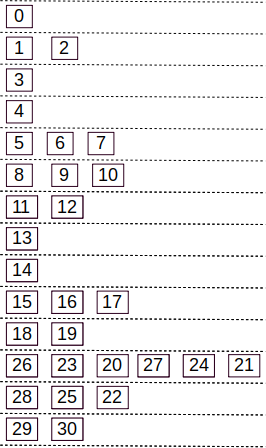


Figure 19: Scheduling

vertex represents an operation within a basic block (shown as rectangles or trapezoids in Figure 18). An operation vertex that controls the flow of operations is called control operations (shown as trapezoids in Figure 18). We assign an unique ID to each operation vertex. There are two types of edges in CDFG: data and control edge. A data edge indicates a data flow, and a control edge indicates a control flow. Data and control edges are shown as solid and dashed arrows respectively. Each basic block preceding a *φ* operation also contains a predicate. For instance, Figure 18 shows that *BB*4, *BB*5 and *BB*6 have predicates. A basic block with a predicate will execute only if the predicate is true. Finally, C2RTL generates a MUX (multiplexer) tree for the *φ* function based on the predicates of the preceding basic blocks (details are discussed later in Section 7.4). For instance, C2RTL converts *< L*4 *> φ* operation in Figure 17b into two MUX operations in Figure 18. Here, the output of the control operations (e.g. *ifout*6 and *ifout*16) would be used as the selectors. Note that, a CDFG is analogous to a logic circuit. Here, each operation can be implemented as a combinational13 or sequential14 logic circuit. In order to implement pipelining, we also need to insert registers among the logic elements. A register holds the output of a logic circuit to be used in the next clock cycle (in the next pipeline stage). Thus, we also need to perform scheduling of each operation in order to divide them among the pipeline stages.

**Step 3: Scheduling.** Scheduling is considered as one of the most critical steps of high-level

13https://en.wikipedia.org/wiki/Combinational logic

14https://en.wikipedia.org/wiki/Sequential logic

synthesis. C2RTL first uses topological sorting to find the order of the operations. It then performs *as soon as possible (ASAP)* scheduling algorithm [77] to schedule each operation in CDFG. C2RTL the calculates the timing slack for each operations (slack indicates how much time an operation can be delayed such that it does not impact the following operation). Finally, it performs *as late as possible (ALAP)* scheduling algorithm [77]. Here, we obtained the latencies of the operations from Bambu’s [133] 45 nm Nandgate Open Cell library characterization. We use the same library for physical synthesis. Our target frequency is 1 GHz (i.e. each cycle is 1 ns). Figure 19 shows the scheduling results for the CDFG in Figure 18. It shows that the program needs 14 clock cycles (14 pipeline stages). Here, the operations scheduled in the same clock cycle will be executed in the same pipeline stage.

**Step 4. Register Insertion.** In a pipelined ASIC, if the result of an operation is used by an another operation scheduled in a different clock cycle, we need to insert a register after the former operation to hold its result. For instance, Figure 18 shows that the output of operation 2 is used as the input to operation 3. However, Figure 19 shows that operation 2 and 3 are scheduled in different clock cycles. In this case, we need to insert a register after operation 2 to store its value so that operation 3 can use it in the next clock cycle. C2RTL inserts one or multiple registers after each operation where the result of the operation crosses the cycle boundary. It also inserts a register after the output to hold the result.

**Step 5: RTL generation.** This step generates synthesizable15 Verilog RTL from the CDFG. We implement each operation of CDFG using a verilog module from a component library. Here, we adapted the component library from Bambu [133]. The component library consists of synthesiz- able Verilog modules that implements primitive components such as arithmetic operations, bitwise operations, MUX, register, SRAM and so on. The *arrays* in the input program are implemented as a multiport register file or SRAM. This is how, C2RTL generates synthesizable Verilog code corresponding to the CDFG. C2RTL also generates a verilog testbench for the input program.

# Resource Binding

Resource binding is a critical step of a high-level synthesis tool [41, 53]. However, that step was not needed in our case. Here we are going to explain the reason. A component library may have

15Verilog has many language constructs which are not synthesizable (i.e. cannot be used to generate physical chip layout). We do not use them here.

multiple resources to implement an operation. There are area, power and timing tradeoffs among the resources [104]. A high-level synthesis tool has to perform resource selection such that the timing requirements are met while the area and power consumption are minimized. This process is known as resource binding. Modern HLS tools perform resource binding during scheduling (instead of two separate steps namely resource allocation and resource binding) [104]. However, our component library has only one resource/module for each operation, thus we did not need to perform resource binding in this dissertation.

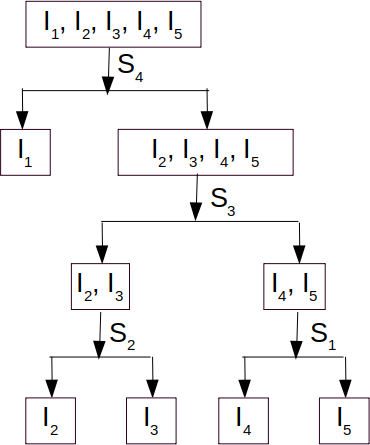
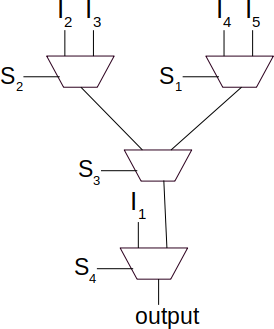
Table 8: An example predicates for a SSA *φ* operation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Selectors | | | |
| Inputs | *S*1 | *S*2 | *S*3 | *S*4 |
| *I*1 | ∗ | ∗ | ∗ | 0 |
| *I*2 | ∗ | 1 | 1 | 1 |
| *I*3 | ∗ | 0 | 1 | 1 |
| *I*4 | 0 | ∗ | 0 | 1 |
| *I*5 | 1 | ∗ | 0 | 1 |

# MUX Tree Generation

An SSA *φ* operation selects one of the inputs based on the predicates (as discussed in Section 7.3). A predicate consists of multiple selectors and an input. Table 8 shows example predicates of a *φ* operation. Here, the *φ* operation needs to select one of the 5 inputs based on the 4 selectors. The value of a selector can be 0, 1 or ∗ (*don’t care*). For instance, if *S*1 = 1, *S*2 = 0, *S*3 = 1 and *S*4 = 1, then *φ* operation will select *I*3. We need to create a MUX tree to implement the *φ* operation. Our component library only contains a 2-to-1 MUX. This is why, we construct a MUX tree where each node is a 2-to-1 MUX.

In order to generate a MUX tree, we first create a predicate splitting tree as shown in Figure 20a. The tree is constructed as follows. The root node will contain all the predicates. We then find a selector for which the value would be either 0 or 1 and not ∗ (*S*4 in this case). We then split the

(a) Predicates splitting tree (b) Corresponding MUX tree

Figure 20: MUX tree generation

root node by the selector. It divides the predicates into two groups ({*I*1} and {*I*2, *I*3, *I*4, *I*5}). We split {*I*2, *I*3, *I*4, *I*5} further based on a selector for which the value would be 0 or 1 and not ∗ (*S*3 in this case). Thus, *S*3 will split the node into two groups: {*I*2, *I*3} and {*I*4, *I*5}. This is how, we split each node of the tree until each node has exactly one predicate. We split a node based on a selector for which the value is either 0 or 1 and not ∗.

Figure 20b shows the actual MUX tree corresponding to the predicate splitting tree in Figure 20a. Each split in the predicate splitting tree is replaced by a MUX.

# Evaluations

We evaluate C2RTL by several IP lookup and packet classification algorithms. We implemented SAIL [184], Poptrie [7] and CP-Trie [86] based IPv6 lookup algorithms and TabTree [117] based packet classification algorithm. Our implementation would be found in C2RTL repository. We generate Verilog RTL from the algorithm implementation using C2RTL. C2RTL uses correct-by- construction approach to generate Verilog RTL code (similar to most other HLS tools including Bambu [133], Catapult [53] and so on). We still evaluate the generated Verilog RTL codes using Icarus Verilog[78]. Our Verilog simulation shows that our generated Verilog RTL produces correct

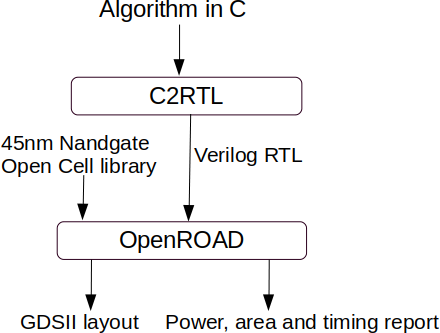


Figure 21: C code to physical chip layout generation

output.

# Physical Synthesis

We evaluate our generated RTL code by an electronic design automation (EDA) tool. EDA tools such as OpenROAD [1], OpenLANE [63] and Synopsis Fusion compiler [161] enable us to generate physical chip layout directly from Verilog RTL code. Here, we use OpenROAD. OpenROAD takes Verilog RTL code, macros (e.g. SRAM, I/O ports, etc), a cell library and constraints (e.g. clock cycle) as inputs and generates corresponding physical chip layout in GDSII format. Figure 21 shows our C code to physical chip layout generation mechanism. Here, OpenROAD takes Verilog RTL code and standard cell library as inputs and generates physical chip layout in GDSII format. OpenROAD also produces timing, area and power characteristics of the Verilog code. Here, we use CMOS 45 nm Nandgate Open Cell Library in OpenROAD.

Our OpenROAD timing report shows that our implementation of SAIL, Poptrie, CP-Trie and TabTree can achieve 1 GHz without timing violation. That is, our generated RTL of pipelined ASIC can lookup or classify 1 packet in every 1 ns. Thus, they can process 1 billion packets per second.

Note that, currently there is no SRAM available for 45 nm Nandgate cell library, to the best of our knowledge. This is why, we use registers to implement the array. Table 9 shows the summery of OpenROAD report for SAIL, Poptrie, CP-Trie and TabTree. This shows that C2RTL can be

Table 9: Power, area and timing in ASIC

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Poptrie | SAIL | CP-Trie | TabTree |
| Clock speed | 1 GHz | 1 GHz | 1 GHz | 1 GHz |
| Internal Power | 76*.*5 mW | 0*.*722 mW | 64*.*6 mW | 0*.*033 mW |
| Switching Power | 24*.*4 mW | 0*.*229 mW | 22*.*2 mW | 0*.*0054 mW |
| Leakage Power | 1*.*15 mW | 0*.*0108 mW | 0*.*926 mW | 0*.*00061 mW |
| Total Power | 102*.*05 mW | 0*.*961 mW | 87*.*726 mW | 0*.*0391 mW |
| Area | 0*.*0658 *mm*2 | 0*.*00061 *mm*2 | 0*.*0523 *mm*2 | 0*.*000034 *mm*2 |

instrumental in evaluating different IP lookup and packet classification algorithms at system level.

# Discussion

C2RTL is aimed to generate various intellectual property (IP) blocks for routing ASICs. However, more works need to be done to achieve the goal.

Currently, C2RTL works for trie based IP lookup and packet classification algorithms. We also need to support other dataplane functions, such as, ethernet lookup [100], OpenFlow packet classification [75], queuing [150], router-assisted congestion control [82, 83], heavy-hitter detection [149], hash based IP lookup and packet classification algorithms [135, 118] and so on.

The QoRs of HLS-generated RTL code often lag behind those of manual RTL code [166, 109]. Research should be done to compare the QoRs of C2RTL to those of manually written RTL. Currently, C2RTL depends on GCC to optimize the code. However, it has shown that a custom compiler flow can improve the QoRs of HLS [112]. A research should be done on improving QoRs of C2RTL.

Finally, the IPs generated by C2RTL needs to be integrated with other IPs in a routing SoC. Research should be done on the integration of the IPs. We also need to tape-out the resulting SoC to validate our flow.

# Conclusion

This dissertation presents a high-level synthesis tool named C2RTL. C2RTL takes an IP lookup or packet classification algorithm (in C) as input and generates corresponding Verilog RTL for pipelined ASIC. Our evaluation shows that the Verilog RTL produced by C2RTL works properly in simulation and physical synthesis.

# CHAPTER 8

**Using Cumulative PopCount to improve SAIL based IPv4 Lookup in Linux Kernel and Domino based programmable ASIC**

This chapter presents a cumulative popcount based IPv4 lookup in Linux Kernel and Domino based programmable ASIC. Here, IP lookup is performed using SAIL [184] algorithm. SAIL consumes very large memory. Here we use *cumulative PopCount* in SAIL that reduces the memory consumption of SAIL by up to 80%. We have implemented SAIL in Linux kernel. We also have implemented SAIL using Domino programming language. Our implementation shows that a programmable pipeline can execute SAIL at line rate. We have evaluated our implementation with FIBs from real backbone routers. Our experimental results show that SAIL with *population counting* is suitable for implementing both dataplane and control plane of a high-speed router.

# FIB Lookup in IPv4

FIB lookup is a key function of a router. A router needs to extract the destination IP address of an incoming packet and find the outgoing port based on the longest prefix match (LPM) algorithm. For instance, Table 10 shows an example FIB table of a router. If the destination IP address of an incoming packet is 169*.*254*.*198*.*1, then according to the longest prefix match algorithm, the packet should be forwarded to *eth*4. Again if the destination IP address of an incoming packet is 169*.*254*.*190*.*5, then the packet should be forwarded to *eth*3. The number of routes in the FIB of a backbone router already has exceeded 750*K* and is growing rapidly [17]. A high-speed router needs to perform around 1 billion FIB lookups per second in order to sustain the line rate [20]. Performing FIB lookup at such a high rate in such a large FIB table is particularly challenging.

Recently several data structures have been proposed that exhibit impressive lookup perfor- mance. These include SAIL [184], Poptrie [7] and DXR [186]. Here SAIL outperforms16 Poptrie

16We assumed that the FIBs are stored in the on-chip memory. SAIL would perform worse if the FIB is partially stored in the off-chip DRAM.

and DXR [7]. A major drawback of SAIL is its high memory consumption. For instance, it con- sumes 29*.*22 MB for our example FIB table with 760*,*195 routes. This often exceeds the on-chip memory of the lookup engine. This dissertation utilizes *population counting* [176] which reduces the memory consumption of SAIL by up to 80%. For instance, SAIL consumes only 6*.*09 MB for the same routing table when *population counting* is used. It is noteworthy that population counting has also been utilized by Poptrie. However, population counting mechanism of Poptrie cannot be applied in SAIL directly (discussed in Section III). SAIL has two variants: SAIL L and SAIL U. SAIL L is optimized for FIB lookup and SAIL U is optimized for FIB update. It is noteworthy that both SAIL L and SAIL U consumes the same amount of memory. *Population counting* reduces the same amount of memory for both SAIL L and SAIL U.

Table 10: Routing table (also known as FIB table)

|  |  |
| --- | --- |
| Prefix | Outgoing port |
| 10*.*18*.*0*.*0*/*22 | *eth*1 |
| 131*.*123*.*252*.*42*/*32 | *eth*2 |
| 169*.*254*.*0*.*0*/*16 | *eth*3 |
| 169*.*254*.*192*.*0*/*18 | *eth*4 |
| 192*.*168*.*122*.*0*/*24 | *eth*5 |

We have implemented both SAIL L and SAIL U in Linux kernel. Our implementation uses *population counting* to reduce their memory consumption. Our implementation enables us to perform FIB lookup, FIB update, FIB delete and FIB flush. It is noteworthy that SAIL on a general purpose CPU cannot sustain the line rate needed for a high-speed router. A high speed router often uses TCAM for implementing FIB lookup. TCAMs however are very power hungry and very limited in capacity [7]. This is why, commercial router vendors often implement FIB lookup using hardware ASIC [50]. Hardware ASICs however are not programmable. Recently programmable pipeline based VLIW processors have emerged as an alternative to a hardware ASIC [149, 20]. Example of such programmable pipeline based routing/switching chips include Barefoot Tofino [168], Cavium Xplaint [181] and so on. They allow us to program the router

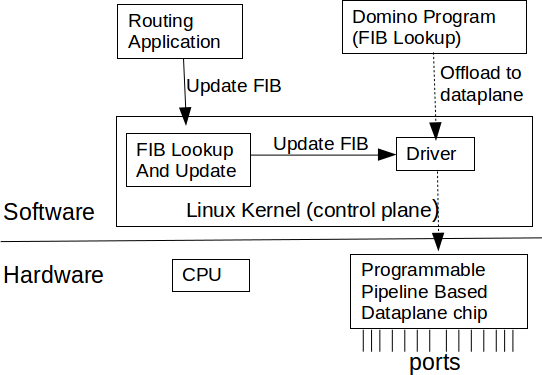
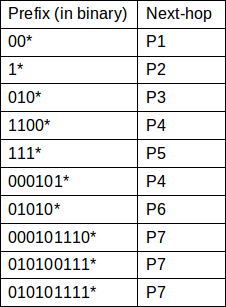
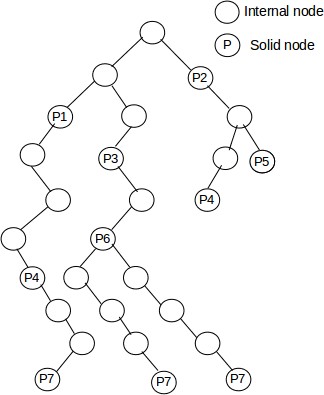
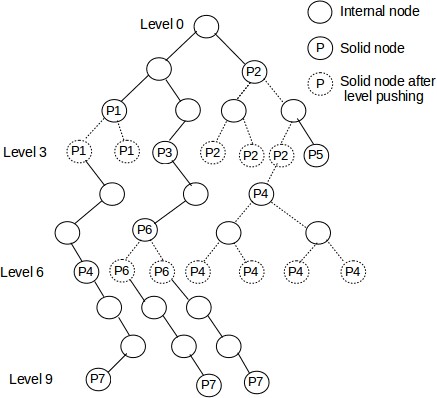


Figure 22: Programmable pipeline based Linux router

dataplane while achieving very high throughput. For instance, Barefoot Tofino 2 programmable routing chip can achieve upto 12*.*8 Tb/s aggregate throughput. Sivaraman et al. has developed Domino programming language to develop programs for programmable pipelines [149]. 17 Figure 22 shows a block diagram of a Linux router where the dataplane is implemented by a programmable pipeline. Here Linux kernel is responsible for managing the FIB table. An user space routing application (e.g. Quagga) is responsible for implementing the routing protocols. Here the FIB lookup is performed by the programmable pipeline. The programmable pipeline runs a Domino program that implements the FIB lookup. When the FIB in the kernel is updated by the userspace application, kernel uses the driver to update the SAIL data structures in the programmable pipeline. It is noteworthy that Cumulus Network also uses Linux in their switches [42]. Currently however they only support fixed-function switching chips which are not programmable.

Currently there is no programmable pipeline chip that supports Domino programming language. We evaluate our Domino based FIB lookup using Domino compiler. Domino compiler enables us to evaluate a Domino program without needing an actual hardware. Domino compiler follows *all- or-nothing approach*. Domino-compiler compiles a domino program if the program can run in a programmable pipeline at line-rate. If the program cannot run at line-rate, the compilation fails. We have implemented SAIL based FIB lookup using Domino programming language. Domino compiler shows that the FIB lookup of SAIL U and SAIL L require 15 and 32 stage pipeline respectively. It

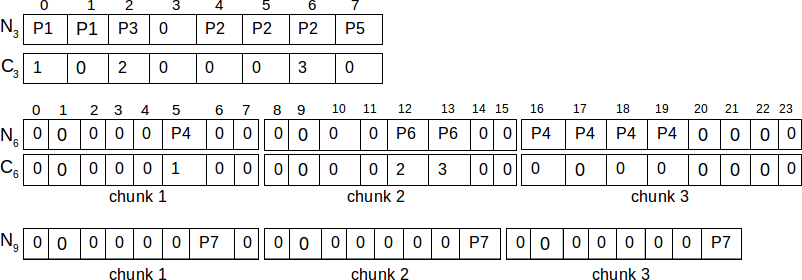
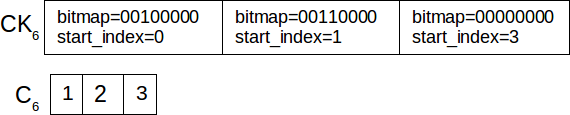
17This is a hardware model. The implemention of the chip does not exist.

* + 1. FIB (b) Binary tree corresponding to (c) Tree constructed by SAIL U. Here solid nodes in

the FIB

level 1 − 2 are pushed to level 3; solid nodes in level 4 − 5 are pushed to level 6; solid nodes in level 7 − 8 are pushed to level 9

(d) *N* and *C* array are constructed based on the nodes in level (e) *C*6 in Figure 23d is encoded with bitmap

3*,* 6 and 9 [184].

and a revised *C*6 where all the zero entries are eliminated

Figure 23: SAIL U with *population counting*

also shows that a 1 GHz VLIW processor can perform 1 billion FIB lookup using SAIL.

# Population Counting

This section discusses how *population counting* is used in SAIL. For the sake of completeness, we describe SAIL based FIB lookup at first. We then show how *population counting* is applied in SAIL.

SAIL splits a routing table into three levels: level 16*,* 24 and 32. It then represents those levels with arrays. Here we explain the approach with a simpler example where the routing table is split

into three levels: level 3*,* 6 and 9. Figure 23a shows an example FIB. Note that different prefixes in a FIB can have the same next-hop. For instance, next-hop *P* 4 and *P* 7 appeared more than once in the FIB. The FIB is then represented by a binary tree as in Figure 23b. Here all the nodes that have a next hop are called solid nodes. All the nodes that do not have a next hop are called internal nodes. Figure 23c shows the tree constructed by SAIL U. SAIL U pushes all the solid nodes in level 1 − 2 are pushed to level 3; all the solid nodes in level 4 − 5 are pushed to level 6, and the

solid nodes in level 7 − 8 are pushed to level 9. It then represents the level 3, 6 and 9 with nexthop

(*N* ) and chunk ID array (*C*). Let us use *Ni* and *Ci* to represent *N* and *C* of level *i*. There can be at most 8(23) nodes in level 3 in Figure 23c. This is why, the size of both *N*3 and *C*3 is 8. *N*3 and *C*3 are constructed based on the solid nodes in level 3 of Figure 23c. *N*3 contains the next-hops in level 3. On the other hand, *C*3[*i*] indicates if there is a longer prefix than the prefix corresponding *N*3[*i*]. *C*3[*i*] = 0 indicates that there is no longer prefix exist. Again *C*3[*i*] *>* 0 indicates that there is a longer prefix and the prefix will be found in chunk *C*3[*i*] of level 6. As the size of a chunk is fixed (23 in this case), chunk ID enables us to locate the longer prefix in the *N*6.

Let us assume that we want to lookup an IP address 11001011*b* in the routing table. It proceeds as follows: It extracts 0-th to 2-th bit (three leftmost bit) from the IP address (6 in this case). *N*3[6] = *P* 2 indicates that *P* 2 is a potential next-hop for this IP address. However, *C*3[6] = 3 indicates that there might be a longer prefix for the IP address. The longer prefix will be found in chunk 3 of level 6. SAIL U then extracts 3-th bit to 5-th bit from the IP address (2 in this case). This is why, the longer prefix will be found in *N*6[(3 − 1) ∗ 8 + 2]. *N*6[18] = *P* 4 indicates that *P* 4 is

the potential next-hop for the IP address. Again *C*6[18] = 0 indicates that there is no longer-prefix

exists for this IP address, thus *P* 4 is the actual next-hop.

**Population counting on** *C*6**:** *C*6 in Figure 23d contains the chunk IDs for level 9. However, most of the entries of the array remain 0 in practice. Here we use bitmap to eliminate all the zero entries from *C*6. This is known as population counting which was described in Warren. et al. [176]. Figure 23e shows how *C*6 in Figure 23d is compressed with the aid of a chunk array *CK*6. Here *CK*6 contains a bitmap and a start index for each chunk. Bitmap encodes the indices which are not zero in the chunk. Start index on the other hand shows where the chunk starts from in the revised *C*6 (in Figure 23e). For instance, *CK*6[1]*.bitmap* = 00110000 indicates that chunk 2 contains two elements and the elements are at index 4 and 5 inside the chunk. Again, *CK*6[1]*.start index* = 1

indicates that chunk 2 start from *C*6[1] in the revised *C*6 in Figure 23e.

It is noteworthy that Poptrie [7] also uses population count. It uses a 64-bit bitmap. Thus, Poptrie can process 6 bits (64 = 26) in every step of the way. Poptrie splits a FIB into four levels: level 16*,* 22*,* 28 and 34. SAIL, on the other hand, splits a FIB into three levels: 16*,* 24 and 32. This is why, it needs to use a 256-bit bitmap so that it can process 8 bits in every step of the way. This is the reason Poptrie cannot be applied directly to SAIL.

It is noteworthy that although a 256-bit bitmap may seem to be a trivial change, it however requires extra processing for both FIB lookup and FIB update. FIB lookup uses POPCNT [176] CPU instruction which can process only 64 bits at a time (on a 64-bit computer). This is why, we divide the bitmap into four parts. Listing 1 shows how the bitmap (*chunk* structure) is divided into four parts. Here each part contains a 64-bit bitmap and an offset to *C* array for that part. Our SAIL based FIB lookup extracts 8 bits from the IP address in each step and maps it to a part of the *chunk* structure (instead of the whole chunk itself). This is how, we process 8 bits in each step of the way while invoking POPCNT only once in each step. The details of the lookup processing and mapping to a part will be discussed in the next section.

**struct** chunk {

Listing 8.1: Chunk structure

u64 bitmap [ 4 ] ; *//256*− *b i t bitmap .*

*// Index to C24 where the chunk i s s t a r t e d .*

u64 s t a r t i n d e x [ 4 ] ;

};

Table 11: Data structures used in SAIL

|  |  |
| --- | --- |
| *def nh* | Default next-hop (level 0) |
| *N*16 | Next-hop array for level 16 |
| *C*16 | Chunk ID array for level 16 |
| *N*24 | Next-hop array for level 24 |
| *CK*24 | Chunk array for level 24 |
| *C*24 | Chunk ID array for level 24 where zero entries are eliminated |
| *N*32 | Next-hop array for level 32 |
| *CK SZ* | Chunk Size (256 in this case) |

# Lookup Algorithm

This section describes SAIL U based FIB lookup where *population counting* is used. An IPv4 address is 32 bit long. Thus, a prefix can be up to 32 bit long. SAIL U represents a FIB as in Figure 23c. It however pushes all the solid nodes in level 1 − 15 to level 16; the solid nodes in level 17 − 23 are pushed to level 24; and the solid nodes in level 25 − 31 are pushed to level 32. Table 11 shows the data structure used by SAIL U. It is noteworthy that there is no *C*32. As level 32 is the last level, it does not need *C*32. Here the size of both *N*16 and *C*16 is 65536 (216). Again the size of each chunk in level 24 and 32 is 256 (28). *CK*24 is a chunk array where each chunk consists of a bitmap and start index. As the size of each chunk is 256, the size of the bitmap needs to be

256. The chunk structure is shown in Listing 1. Note that instead of having just one *start index* for the whole chunk, we divide the chunk into 4 parts and save the *start index* for each part. FIB lookup maps an IP address to a part of the chunk during the lookup.

Algorithm 1 shows the lookup process of SAIL U where population counting is used. It starts by initializing the *nexthop* to the default route (Line 2). It then extracts 16 bits from the LSB of the IP address. This will be used as the index to *N*16 and *C*16. If *N*16[*i*] *>* 0, SAIL U will update the *nexthop* (Line 5-6). If *C*16 contains a valid chunk ID for the IP address, then SAIL U will further calculate the index to *N*24; otherwise it will return the *nexthop* (Line 7-13). SAIL U then checks if *N*24[*i*] *>* 0; if yes, then it updates the *nexthop* (line 14-15). It now needs to check if there

**Algorithm 2** SAIL U based FIB Lookup with *population counting*

**Input**: Destination IP address of the packet - *ip*

**Output**: Next-hop - *nexthop*

1: **procedure** Lookup(*ip*)

2: *nexthop* ← *def nh*

3: /\*Extract 16 bits from *ip* \*/

4: *idx* ← *ip >>* 16

5: **if** *N*16[*idx*] *>* 0 **then** 6: *nexthop* ← *N*16[*idx*] 7: **if** *C*16[*idx*] ! = 0 **then**

8: /\*Extract bit 17-24 from *ip* \*/

9: *ck off* ← ((*ip* & 65280) *>>* 8)

10: *ck id* ← *C*16[*idx*]

11: *idx*24 ← (*ck id* − 1) ∗ *CK SZ* + *ck off*

12: **else**

13: return *nexthop*

14: **if** *N*24[*idx*24] *>* 0 **then**

15: *nexthop* ← *N*24[*idx*24]

16: /\*Check if there is a chunk in level 32 for this IP\*/

17: *part idx* ← *ck off*

64

18: *part off* ← *ck off* % 64

19: *bitmap* ← *CK*24[*ck id* − 1]*.bitmap*[*part idx*] 20: *idx* ← *CK*24[*ck id* − 1]*.start index*[*part idx*] 21: **if** *bitmap* & (1*ULL << part off* ) **then**

22: /\*Calculate the index to *C*24\*/

23: *c*24*i* ← *idx* + *BITCOUNT* (*bitmap, part off* )

24: /\*Extract 8 bits from the MSB of *ip* and calculate index to *N*32\*/

25: *idx*32 ← (*C*24[*c*24*i*] − 1) ∗ *CK SZ* + *ip* & 255

26: **else**

27: return *nexthop*

63

28: **if** *N*32[*idx*32] *>* 0 **then**

29: *nexthop* ← *N*32[*idx*32]

is a chunk in level 32 for this IP address. It does so by checking the bitmap. Note that a chunk (in Listing 1) is divided into 4 parts. SAIL U first finds the part and the offset inside the part for the IP address by a simple division and a modulo operation (Line 17 − 18). 18 Note that although it

may look very trivial, this approach allows us to use population counting while processing 8 bits

(instead of 6 bits) in every step of the way. Moreover, this approach can further be extended to where FIB lookup can process more than 8 bits in every step of the way. For instance, if we want to develop a FIB lookup that processes 12 bits in every step of the way, we can do so by maintaining a 4096-bit bitmap and dividing the bitmap into 64 parts. Such an approach would be beneficial for lookup in IPv6 FIB where each prefix can be up to 128 bit long. Line 19 − 20 gets the bitmap and the start index for the IP address. Line 21 checks if there is a chunk for this IP address in level 32; if yes, it then calculates the index to *C*24 (Line 23). Note that it uses POPCNT instruction

[176] to count the number of bits set to 1. Most of the modern *X*86 processor supports POPCNT instruction. This allows us to convert the bitmap to an offset. Finally it calculates the index to *N*32 in Line 25. It is noteworthy that SAIL U does not need to backtrack in order to find the longest-prefix match. It visits level 16, 24 and 32 in order and updates *nexthop* in each step. This is why, *nexthop* will contain the next-hop corresponding to the longest matching prefix.

Note that population counting is also applicable to SAIL L in the similar manner. In our implementation, we have used population counting in both SAIL U and SAIL L. Our FIB update mechanism also populate the *CK*24 array properly so that it can be utilized by FIB lookup. The details of our FIB update mechanism would be found in our implementation.

# Implementation

We have implemented both SAIL U and SAIL L in Linux kernel. Our implementation contains 1195 and 1300 lines of *C* code respectively. We have made the source code publicly available as Linux kernel patches. 19 20 Our website21 also contains the instructions to include the patches into Linux kernel source code. These patches have been produced with Linux kernel 4*.*19*.*0-*rc*6+. Our implementation uses population counting to reduce the size of *C*24. It is noteworthy that in actual implementation, *N*16, *N*24 and *N*32 contain next-hop indices instead of actual next-hops.

18The modulo and division operations can be replaced by bitwise operations 19https://github.com/tamimcse/kernel-patches/blob/master/sail-u.patch 20https://github.com/tamimcse/kernel-patches/blob/master/sail-l.patch 21https://web.cs.kent.edu/~mislam4/sail-in-linux-kernel.html

This allows us to use just one byte to store the next-hop index instead of using an actual pointer which is 8 bytes in a 64-bit machine. Note that, this approach is only applicable when the router has up to 255 ports. If a router has more than 255 ports, it will take 2 bytes to represent each next-hop index.

It is also noteworthy that although we use *level-pushing*, we still need to store the actual prefix length in order to support incremental updates. This is why it maintains an array *P*16 which stores the actual prefix length of each prefix in *N*16. In the same way, it maintains arrays *P*24 and *P*32 which store actual prefix lengths corresponding to prefixes in *N*24 and *N*32 respectively. *P*16, *P*24 and *P*32 are only used for route insertion and route deletion. They are not used in FIB lookup. The details of the route insertion and route deletion will be found in our implementation.

We have tested our implementation with several very large routing tables containing more than 700*K* routes. Here we have tested our implementation by looking up all possible 232 IPv4 addresses.

# 8.4.1 SAIL in a Programmable Pipeline

This section discusses the implementation of SAIL U and SAIL L for a programmable pipeline. The programmable pipeline works as the dataplane in a router. It is noteworthy that the dataplane is only responsible for performing the FIB lookup. FIB update will be performed by the control plane (Linux kernel in this case). The control plane is responsible for computing the SAIL arrays and updating the corresponding data structures in the dataplane. The dataplane will simply perform the FIB lookup based on the arrays.

We have developed both SAIL U and SAIL L based FIB lookup which contain 67 and 70 lines of Domino code respectively. We have made our implementation publicly available. 22 23 Note that we use *popcnt* intrinsic function which counts the number of bits set to 1. This function needs to be supported by the underlying pipeline hardware. Here we use Domino’s *raw* [149] atom to compile our programs. A Domino program compiled by Domino compiler is guaranteed to run at line rate. If a Domino program cannot run on a programmable pipeline, the compilation fails. Domino compiler can compile our implementation. This is why, our implementations are guaranteed to run at line rate.

22https://github.com/tamimcse/domino-examples/blob/master/domino-programs/sail-u.c

23https://github.com/tamimcse/domino-examples/blob/master/domino-programs/sail-l.c

Table 12: Comparison between SAIL U and SAIL L in a Programmable Pipeline

|  |  |  |
| --- | --- | --- |
|  | SAIL U | SAIL L |
| Number of pipeline stages | 15 | 32 |
| Maximum # of atoms/stage | 5 | 6 |
| Processing latency | 15 ns | 32 ns |

# Comparison between SAIL U and SAIL L in a Programmable Pipeline

Currently there is no Domino based programmable pipeline hardware, to the best of our knowledge. Thus, it is impossible to do experimental evaluation at this point. Here we evaluate the performance of SAIL U and SAIL L using Domino compiler. Domino compiler compiles a Domino program and produces a programmable pipeline which would be able to run the Domino program at line rate. The output of our Domino program shows that SAIL U would be able to run on a 15-stage pipeline where the maximum number of atoms-per-stage [149] is 5. On the other hand, SAIL L requires a 32-stage pipeline where the maximum number of atoms-per-stage is 6. We also have made the pipelines publicly available. 24 25 The pipelines show how SAIL U and SAIL L are executed in a programmable pipeline. The results indicate that SAIL U requires fewer pipeline stages (i.e. fewer hardware resources) compared to that of SAIL L.

It is noteworthy that each stage of a pipeline processes one packet in each clock cycle. If each pipeline stage runs at 1 GHz, then each pipeline stage takes 1 ns to process each packet. As SAIL U requires a 15-stage pipeline, the entire pipeline will require 15 ns to perform each FIB lookup. In the same way, SAIL L will require 32 ns to perform each FIB lookup (due to 32-stage pipeline). This is why, although SAIL L performs better than SAIL U in FIB lookup on a general purpose processor, SAIL U based FIB lookup outperforms SAIL L in a programmable pipeline.

Table 12 summarizes the comparison between SAIL U and SAIL L in a programmable pipeline. Note that although the pipelines of SAIL U and SAIL L require different numbers of stages, they both process one packet in each clock cycle. Thus, both SAIL U and SAIL L can be implemented

24https://raw.githubusercontent.com/tamimcse/domino-examples/master/pipeline-sail-u.png 25https://raw.githubusercontent.com/tamimcse/domino-examples/master/pipeline-sail-l.png

in a programmable pipeline at line rate.

Table 13: FIB dataset

|  |  |  |  |
| --- | --- | --- | --- |
| Name | AS Number | # of prefixes | # of next-hops |
| fib1 | 293 | 759069 | 2 |
| fib2 | 852 | 733378 | 138 |
| fib3 | 19016 | 552285 | 236 |
| fib4 | 19151 | 737125 | 2 |
| fib5 | 23367 | 131336 | 178 |
| fib6 | 32709 | 760195 | 140 |
| fib7 | 53828 | 733192 | 223 |

# EXPERIMENTAL RESULTS

Currently Linux uses LC-trie for routing table lookup. However, the lookup performance of LC-trie is significantly slower than that of SAIL [7, 184]. This is why, this dissertation does not compare the performance of SAIL to that of LC-trie. It rather compares the performance of SAIL U to that of SAIL L. Here we evaluate the SAIL variants with FIBs of real backbone routers.

# Data Set

We have obtained the FIBs of real backbone routers from Route Views projects [142]. The FIB snapshot was taken on April. 1, 2019 at 10 PM EST. We have made the FIBs publicly available

26. Table 13 shows the description of the FIB tables. It is noteworthy that Route View project provides us with RIB (Routing Information Base). We then convert a RIB into a FIB (Forwarding Information Base) by keeping only one next-hop per prefix. Actual lookup is performed on the FIB table. Here we have kept IPv4 prefixes only. Lookup in an IPv6 FIB table is beyond the scope of this dissertation. We also replace the next-hop indices by veth0-veth31. These veths (virtual ethernet in Linux kernel) act as the next-hops in our experiment. The Autonomous System (AS) number of Table 13 represents the ISP of the corresponding FIB. For instance, the table shows that

26https://github.com/tamimcse/parse-bgpdump/

*fib2* and *fib6* belong to AS852 (TELUS Communications, an ISP in Canada) and AS32709 (Joink, an ISP in the US) respectively.

# Experimental Setup

The experiment has been conducted on a laptop running our updated Linux kernel 4*.*19*.*0-*rc*6+. The laptop however does not have many physical network interfaces needed to emulate a backbone router. Here we create 32 virtual network interfaces (*veth* kernel module) and use them as the next-hop of the routing table. The laptop has an Intel(R) Core (TM) i7-4510U processor (2GHz, 4 MB L-3 cache) and 16 GB DRAM. The processor here has two physical cores (i.e. when the *hyper-threading* is disabled). It is noteworthy that *hyper-threading* results in unnecessary cache thrashing. This is why we disable hyper-threading before conducting the experiment. We also disable *frequency-scaling* of the CPU in order to get deterministic results.

Table 14: Impact of population counting on memory consumption (for *fib6* )

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Without Population Counting | | With Population Counting | |
| Array | Length | Size | Length | Size |
| *N*16 | 65536 | 64 KB | 65536 | 64 KB |
| *C*16 | 65536 | 128 KB | 65536 | 128 KB |
| *N*24 | 6071808 | 5*.*79 MB | 6071808 | 5*.*79 MB |
| *CK*24 |  |  | 366 | 22*.*87 KB |
| *C*24 | 6071808 | 23*.*16 MB | 366 | 1*.*42 KB |
| *N*32 | 93696 | 91*.*50 KB | 93696 | 91*.*50 KB |
| Total |  | 29*.*22 MB |  | 6*.*09 MB |

# Memory Consumption

Table 14 shows the impact of population counting on memory consumption. Here we show the memory consumption for *fib6*. Note that, the memory consumption primarily differs for *C*24. The size of *C*24 was 23*.*16 MB before population counting was applied whereas the size is only 1*.*42 KB after population counting is applied. The reason for this is as follows. Most of the prefixes in *fib6*

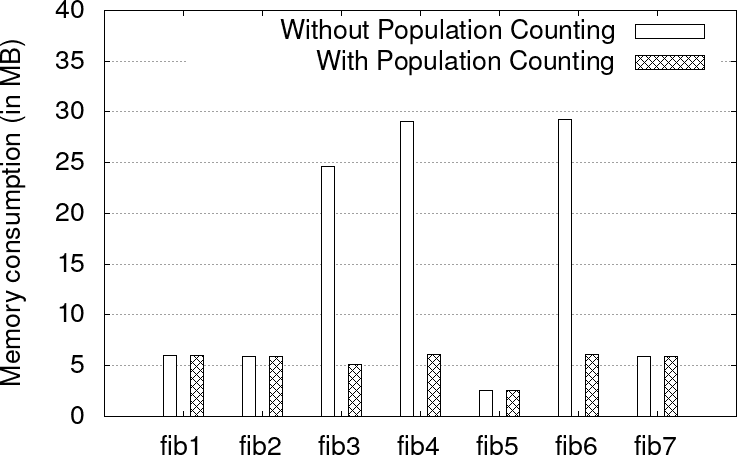


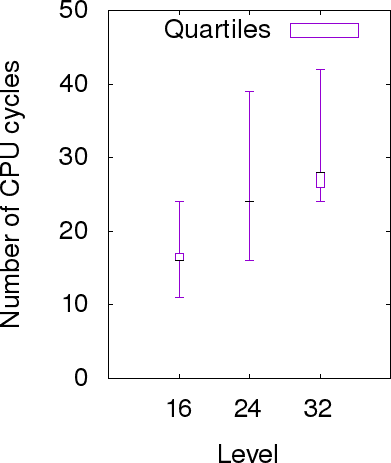
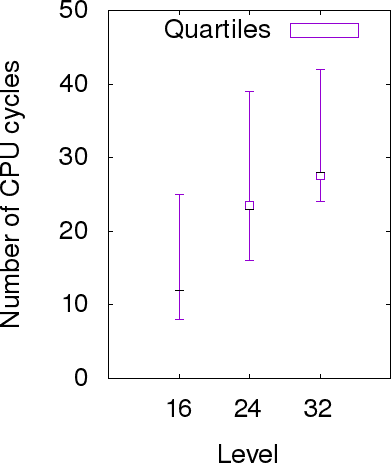
Figure 24: Memory consumption for different FIBs

are with length 0 − 24. It has found that more than 98*.*5% routes in backbone routers are 0 − 24 bit long [17]. There are very few routes with length 25 ∼ 32 in backbone routers. This is because a backbone router is very less likely to have a direct link to an end host. As a result, very few chunks will be needed in level 32. As a result, most of the entries in *C*24 remain zero. Population counting eliminates all the zero entries in *C*24 by having an additional *CK*24. This reduces the memory consumption significantly.

Figure 24 shows the memory consumptions for the FIBs of our data set. It shows that popula- tion counting reduces the memory consumption significantly for some FIBs (*e.g. fib3, fib4, fib6* ). However, if all the prefixes in a FIB are with length 0 − 24, SAIL does not populate *C*24. SAIL does not consume very large memory in that case. Population counting also has no effect on memory consumption in that case (e.g. *fib1, fib2, fib5, fib7* ).

# Lookup Cost

We measure the lookup cost based on the number of CPU cycles needed per lookup. It is noteworthy that measuring the cycle count accurately is not straightforward. We measure the cycle count using RDTSC instruction [179]. But a CPU often reorders the instructions which lead to inaccurate cycle count. To prevent this problem, we also use RDTSCP and CPUID instructions to serialize the instructions. We also disable the interrupts and preemption while measuring the cycle count. Finally we need to deduct the cost of RDTSC instruction itself to get the accurate cycle count of

* + - 1. SAIL U (b) SAIL L

Figure 25: Lookup cost for different levels .

our routing table lookup. We run a tight loop to perform the routing table lookup to ensure that the cache is populated. We also disable hyper-threading and CPU frequency scaling in order to avoid cache thrashing. We have made our test code publicly available as a Linux kernel patch. 27 It shows how to count the number of CPU cycles accurately for each FIB lookup.

The lookup cost of an IP address primarily depends on the level in which the longest matching prefix is found. The lookup cost should be the smallest if the longest prefix is found in level 16. Again the lookup cost should be the highest if the longest prefix is found in level 32. Figure 25 shows the lookup cost of SAIL U and SAIL L for different levels. The figure shows that a general purpose processor cannot achieve deterministic performance. The actual cycle count of a program can vary because of memory/cache access latency, OS scheduler and so on. Note that SAIL L perform slightly better than SAIL U when the longest prefix is found in level 16 or level 24. This is because SAIL U requires one more lookup than SAIL L in that case. If SAIL L finds a prefix in *N*16 or *N*24, it immediately knows that it is the longest prefix. On the other hand, SAIL U needs to check *C*16 or *C*24 to determine if there is a longer prefix. However, as both *C*16 and *C*24 resides in the CPU cache, this cost is very minimal.

27https://github.com/tamimcse/kernel-patches/blob/master/test-fib-lookup.patch

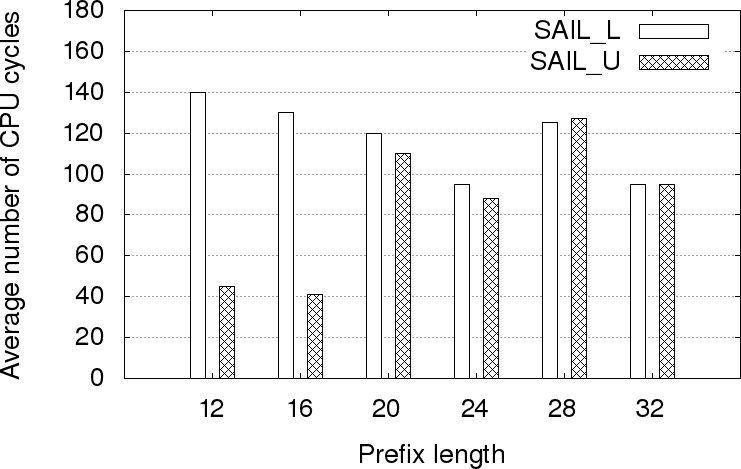


Figure 26: Update cost for different prefix lengths

# Update Cost

Figure 26 shows the average number of CPU cycles needed for FIB update for different prefix lengths. The figure shows that SAIL U performs significantly better than SAIL L for the prefixes with length 0 − 16. SAIL U pushes the prefixes to level 16 whereas SAIL L pushes the prefixes to either level 16, 24 or 32. This is why SAIL L may need to update many more nodes compared to that of SAIL U which increases the update cost. Again SAIL U pushes the prefixes with length 17 − 24 to level 24 whereas SAIL L pushes them to either level 24 or 32. Thus, SAIL U performs similarly or better than SAIL L. A backbone router generally contains very few prefixes in level

32. This is why, most of the prefixes with length 17 − 24 are pushed to level 24 (instead of level 32) in SAIL L. As a result, both SAIL U and SAIL L perform very similarly in most of the cases. However, if a prefix with length 17−24 is pushed to level 32, it increases the update cost significantly which increases the average update cost. Finally Figure 26 shows that both SAIL U and SAIL L performs very similarly for the prefixes with length 25 − 32.

# Conclusion

This dissertation presents an implementation of SAIL (both SAIL U and SAIL L) in Linux kernel and in a programmable pipeline. This is the first implementation of SAIL in those platforms. We use *population counting* to reduce the memory consumption of SAIL. Our experimental results show that it can save the memory consumption by up to 80%. Our implementation shows that

SAIL can be implemented in a programmable pipeline at line rate. It also shows that although SAIL L performs faster FIB lookup compared to SAIL U in a general purpose processor, SAIL U outperforms SAIL L in a programmable pipeline. We have used Domino programming language to develop SAIL for programmable pipelines. However, currently there is no hardware that supports Domino. This is why, we have restricted our experimental evaluation to SAIL in Linux kernel. We have evaluated it with FIBs from real backbone routers. Our experimental results show that our implementation can perform fast FIB lookup and update. It also shows that the FIB lookup in both SAIL U and SAIL L perform quite similarly on a general purpose processor. The FIB update of SAIL U however outperforms that of SAIL L in some cases. Overall it shows that our SAIL implementation can act as a dataplane and a control plane of a high-speed router.

# CHAPTER 9

**Limitations**

We now discuss the limitations of the systems presented in this dissertation. We begin with the limitations that are common to both CP-Trie and C2RTL. We then discuss the limitations of each individual system.

# Limitations common to both CP-Trie and C2RTL project

* + - * **Lack of silicon implementations.** In this dissertation, we evaluated several IP lookup and packet classification algorithms in ASIC. However, none of our algorithms has a hardware implementation in silicon. Designing and testing hardware implementation requires many hardware modules besides IP lookup which are not available to us. Designing a new chip can take a few years and requires significant financial investment and access to a large team of engineers. This is why, we only evaluated IP lookup modules in Verilog simulator (iVerilog) and EDA (OpenROAD). While it is still possible to evaluate the generated Verilog on FPGA, we decided not to do it because FPGA does not accurately reflect a router chip. FPGA runs at much lower speed, consumes much more power and area and have much higher unit price. Note that, there is still value to an FPGA implementation for the following reasons, which we hope to explore in future work.
        + **System Verification.** An FPGA implementation allows us to check the correctness of the Verilog code, especially to test correctness when interfacing a router with the external world.
        + **System Evaluation.** An FPGA implementation can serve as a vehicle for end-to-end evaluation of ideas, much like our Mininet [110] evaluation.
        + **Programmable Network.** The reconfigurability of an FPGA may allow us to develop a programmable router [19, 20, 190].
      * **Lack of SRAM implementations.** Currently there is no SRAM available for 45 nm Nandgate cell library, to the best of our knowledge. Lack of open source standard cell library and memory compiler [48, 67, 9] refrained us from developing SRAM in our ASIC. This is why, we use registers to implement the arrays. Area and power consumption of registers are significantly higher than those of SRAM. Thus, the power and area reported in this dissertation do not accurately reflect the power and area of an actual IP lookup engine. The power and area reported in this dissertation however indicates that CP-Trie will outperform Poptrie in ASIC which is correct.

We even considered using CACTI-P [114] SRAM simulator to calculate the power and area cost of on-chip SRAM needed for Poptrie and CP-Trie. However, the area and power reported by CACTI-P is not very accurate. We also considered using open source memory compilers such as OpenRAM [67] and AMC [9] to generate an on-chip SRAM module. However, cur- rently they use SCN3ME SUBM 0*.*5µm technology which is very old and requires very high power and area, thus does not reflect the area and power consumption of a routing ASIC.

There are many SRAM IP vendors such as Synopsys, Cadence, ARM and so on [67]. In order to develop actual routing ASIC, one of those on-chip SRAM IP needs to be integrated to our compute pipeline. This however will require significant financial investment.

* + - * **Using manufacturable standard Cell library.** Currently, C2RTL uses 45nm Nandgate Standard Cell library. The cell library however is not manufacturable. A research should be done on incorporating open source SkyWater PDK (https://github.com/google/skywater- pdk).
      * **Performance-power-area (PPA) are solely based on OpenROAD.** There is not sim- ple way to measure performance-power-area (PPA) of a chip until performing the physical synthesis of the chip [96]. This dissertation uses OpenROAD to perform physical synthe- sis. Thus, all the performance-power-area (PPA) reports are very specific to OpenROAD. While this is not a limitation of our approach, we should mention that the performance performance-power-area (PPA) in this dissertation are specific to OpenROAD.

# Limitations of CP-Trie

In addition to the common limitations specified in Section 9.0.1, CP-Trie project has the following limitations.

* + - * **Not using real internet traces.** While evaluating IP lookup, we did not use real internet traces. This is because the real Internet traces we obtained [178] were not taken in the same backbone router as the FIBs. Thus, they are not within the ranges of prefixes in FIB. To overcome this problem, we generated synthetic repeated traffics which are analogous to real traffic. Here we randomly selected prefixes from the FIBs (with possible duplicates) and added random bits at the ”don’t care” fields of the prefixes. Each IP in repeated traffic is looked up 10 times repeatedly. Such synthetic repeated traffic is analogous to a router handling a small number of flows. Such repeated traffic is prevalent in real world Internet traces [178].
      * **Software performance is solely measured based on CPU Performance counter (TSC register).** There are several ways to measure software performance. This includes using CPU clock, CPU Performance counter (TSC register)28 and so on. Here, CPU Perfor- mance counter (TSC register) is the preferred way of measuring performance. Here, we have a tight loop to lookup 224 IP addresses and measure the performance using CPU performance counter. Note that, other notable IP lookup algorithms such as Poptrie [7] and SAIL [184] have been measured in the same process. While this is not a limitation of our approach, we should mention that the performance reported in this dissertation is specific to the CPU performance counter.

# Limitations of C2RTL

In addition to the common limitations specified in Section 9.0.1, C2RTL project has the following limitations.

* + - * **Lack of completeness theorems.** We do not have a formal characterization of what can and cannot be done by C2RTL. While we have examples of algorithms that can and cannot be synthesized by C2RTL, it would be ideal to have a “completeness” theorem that states

28https://en.wikipedia.org/wiki/Time Stamp Counter

Table 15: Area and delay trade-offs for an adder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Delay (ps) | 220 | 400 | 580 | 760 | 940 | 1220 |
| Area | 556 | 254 | 225 | 216 | 210 | 206 |

that all the algorithms within a class (and within that class alone) can be synthesized by C2RTL. Such theorems would also give a formal assurance of the system being “future proof” so long as a new algorithm falls within a particular class of algorithms. An example of such a theorem is the equivalence of finite-state machines and regular expressions, which states that any regular expression whether a regular expression that is known today or an unforeseen regular expression can be expressed using a finite-state machine. Note that, most of the existing HLS tools lack such completeness theorems.

* + - * **Lack of formal verification.** C2RTL generates Verilog RTL code from C code. However, there is no formal verification if the C code and the resulting Verilog code have the same functional behavior. Like almost all other HLS tools, C2RTL uses correct-by-construction approach to generate the Verilog RTL code. However, a bug in a HLS tool can cause to gen- erate incorrect Verilog code which can have very serious ramifications. This is why, a formal verification tool or equivalence checker for a HLS tool is needed [108, 73, 113]. Currently, C2RTL does not have an equivalence checker. Note that, most of the existing HLS tools also lack an equivalence checker. To avoid this limitation, existing HLS tools generally use Verilog simulation against a test-bench. We also used a similar approach. We used iVerilog simulator to test our generated Verilog code.
      * **Using a fixed resource for each operation.** Currently, C2RTL mainly focuses on im- plementing pipelined ASIC. It divides the behavioral description into several pipeline stages where each pipeline stage can be executed in 1ns to achieve 1 GHz clock speed. Here, each operation is implemented by a fixed resource (similar to Bambu [133]). This is however a sim- plistic assumption. An instruction can have multiple resource implementations in a standard cell library. And, there is a trade-off between delay and area among multiple implementations.

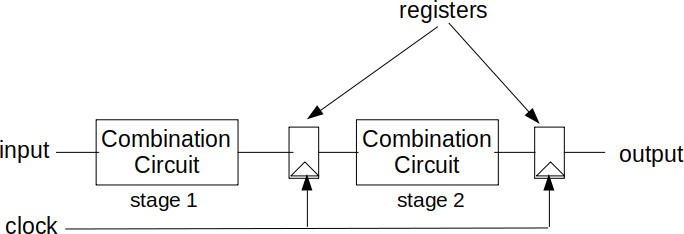


Figure 27: Block diagram of a pipelined ASIC

For instance, Table 15 shows area and delay trade-off for an adder in TSMC 90nm standard cell library [104]. It shows that increasing latency reduces area (and power). Note that, power consumption of a chip is proportional to its area [99]. This is why, we mainly focus on area, but it is applicable to power as well. It is possible to develop a resource allocation algorithm that reduces area without violating timing constraints. Currently, C2RTL does not have a resource allocation algorithm. It uses a fixed resource for each operation. Thus, the results specified in this dissertation can further be refined with a resource allocation algorithm.

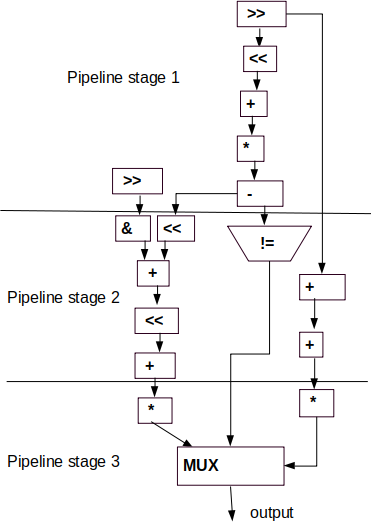
* + - * **Broader data-plane function support.** Currently, C2RTL mainly focuses on trie based routing table lookup and packet classification algorithms. However, it currently does not support many other dataplane functions needed for a router. Those include hash based IP lookup and packet classification, hash based exact match, and so on.

# Future work on Power/Area Efficient Pipelining

This section expands on the limitation of using a fixed resource for each operation. It also describes what can be improved in the future.

# Pipelining

Pipelining involves partitioning a circuit into multiple stages. Figure 27 shows a block diagram of a pipelined ASIC. Here, a circuit is partitioned into multiple pipeline stages. Each pipeline stage is implemented by a combinational circuit. The latency of a combinational circuit has to be smaller than the clock cycle. Output of each combination circuit is stored in a register. The registers are linked to a clock. In every clock cycle, data moves from one pipeline stage to the next stage. Thus, pipelining of circuit is actually a register insertion problem.



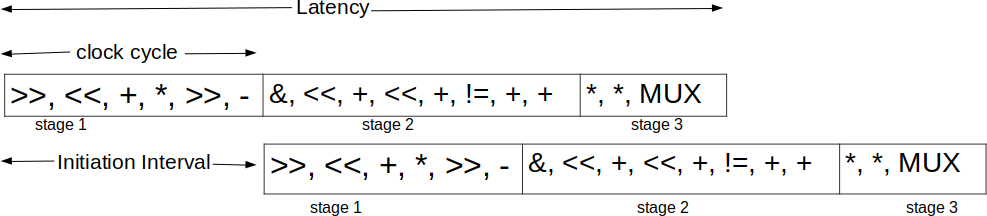
1. Example circuit
2. Execution of a pipeline

Figure 28: Example pipeline with Initiation Interval (II) = 1 and Latency = 3 cycle

# Pipelining terminologies

This subsection describes different terminologies regarding pipelining. There are three key param- eters of a pipeline: 1. **clock cycle**, 2. **Initiation Interval (II)** and 3. **Latency**. Figure 28 shows an example pipeline with Initiation Interval (II) = 1 and Latency = 3 cycle. Here, the circuit is divided into three pipeline stages. Each pipeline stage needs to be executed in a single clock cycle. Thus, the latency of the circuit is the same as the number of clock cycles needed (3 in this case). Here, the initiation interval is 1 which indicates that the pipeline start processing new data in

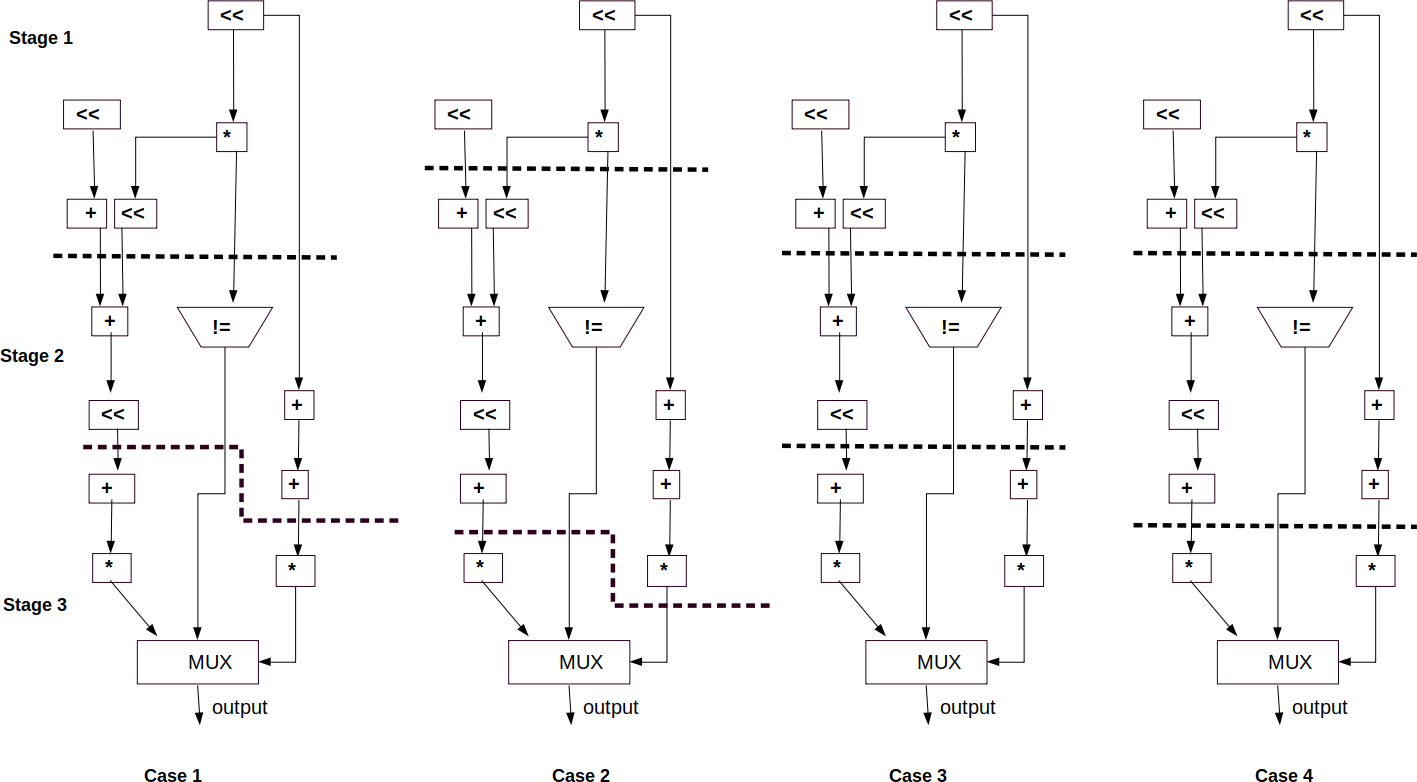


Figure 29: Four arbitrary pipelining options of the same circuit (CDFG)

every clock cycle. In order to process inputs at line rate, the initiation interval needs to be 1. The throughput of the circuit is same as the clock frequency. Thus, if the clock cycle of the pipeline ASIC is 1 ns, the clock frequency is 1 GHz. That is, the circuit can process 109 inputs every second. These terminologies are widely used in existing high-level synthesis systems [53].

A routing chip needs to process packets at line rate. This is why, they use Initiation Interval

(II) = 1. Clock speed of routing ASIC is generally kept at 1 GHz [191], thus the clock cycle is 1 ns.

# Combined Resource Allocation and Scheduling Algorithm

This section describes the problem of **resource allocation** and **scheduling** with an example. Figure 29 shows four **arbitrary** pipelining configurations of a circuit. Here, the latency of the circuit is 3 clock cycle, thus the circuits are divided into 3 pipeline stages. Diving a circuit into multiple pipeline stages is known as **scheduling**. On the other hand, selecting a resource to implement an operation is known as **resource allocation**. Here, the circuit needs adders, multipliers, left shifters, comparators and multiplexers. Table 16 shows an example available resources to implement the operations. Let us assume that the clock cycle is 1 ns. If we use the slowest (and thus most

Table 16: Available resources to implement *adder*, *multiplier*, *left shifter*, *comparator* and *multi- plexer*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Add | Delay (ps) | 220 | 400 | 580 | 810 |
| Area | 556 | 400 | 300 | 150 |
| Mul | Delay (ps) | 250 | 450 | 600 | 850 |
| Area | 810 | 600 | 360 | 195 |
| Left shift | Delay (ps) | 100 | 210 | 250 | 465 |
| Area | 300 | 210 | 100 | 50 |
| Comparator | Delay (ps) | 50 | 140 | 200 | 300 |
| Area | 200 | 150 | 100 | 75 |
| Mux | Delay (ps) | 200 | 250 | 400 | 650 |
| Area | 500 | 400 | 250 | 200 |

area/power efficient) resource, it may violate timing constant. For instance, the case 1 of Figure 29 has two paths in pipeline stage 1. One path contains a left shift and an adder, thus path latency is 810 + 665 (for the slowest resource) which exceeds the clock cycle. Another path contains two left shifters and a multiplier, thus path latency is 665 + 665 + 850 (for the slowest resource) which also exceeds the clock cycle. Thus, simply using the slowest resource may violate the timing constraint. Again, using the fastest resource for each operation may meet the timing constraints, however it will result in a very large area. The objective of the resource allocation algorithm is to find the optimal resource allocation for each operation such that area is minimized without violating the timing constraint. Also note that, we need to consider this for all possible pipelining cases, not just one pipeline configuration. Thus, our **combined resource allocation and scheduling** algorithm needs to find an optimal pipeline configuration and optimal resource assignment such that area is minimized and timing constraint is met.

Table 17: Resource allocation options where path latency is less than 1 ns (latency requirement) for stage 1 in case 1 in Figure 29

|  |  |  |
| --- | --- | --- |
| Resource along with latency | Latency | Area |
| Add(400) + Left-shift (465), Left-shift (210) + Left-shift (100) + Mul (600) | 910 ps | 1320 |
| Add(580) + Left-shift (250), Left-shift (210) + Left-shift (100) + Mul (600) | 910 ps | 1280 |
| Add(810) + Left-shift (100), Left-shift (210) + Left-shift (100) + Mul (600) | 910 ps | 1320 |
| Add(400) + Left-shift (465), Left-shift (250) + Left-shift (250) + Mul (450) | 950 ps | 1250 |
| Add(580) + Left-shift (250), Left-shift (250) + Left-shift (250) + Mul (450) | 950 ps | 1200 |
| Add(810) + Left-shift (100), Left-shift (250) + Left-shift (250) + Mul (450) | 950 ps | 1250 |

Table 18: Optimal resource allocation for case 1 in Figure 29. Here, the path latency of each stage is less than 1 ns, thus latency requirement is met and area in minimized.

|  |  |  |
| --- | --- | --- |
|  | Resource along with latency | Area |
| Stage 1 | Add(580) + Left-shift (250), Left-shift (250) + Left-shift (250) + Mul (450) | 1200 |
| Stage 2 | Add(580) + Left-shift (250), Comparator (300), Add (400) + Add (580) | 1175 |
| Stage 3 | Add(220) + Mul (250) + MUX (250), Mul (600) + MUX (250) | 2126 |
| **Total area** |  | 4501 |

Note that, both scheduling and resource allocation algorithms are mutually dependent. Existing high level synthesis systems implement resource allocation and scheduling sequentially. They first implement a resource allocation algorithm. They then perform a scheduling algorithm. This approach may result in a small area, however it will increase the number of pipeline stages (thus latency). If there is a latency constraint, we need to solve the resource allocation and scheduling together. Table 17 shows the different resource allocation options for stage 1 of case 1 (in Figure 29). All of these resource allocation options meet the timing constraint, however they result in different areas. Here, the minimum area is 1200 which is the optimal resource allocation for that

Table 19: Optimal resource allocation for different pipeline options

|  |  |  |  |
| --- | --- | --- | --- |
| Case 1 | Stage 1 | Add(580) + Left-shift (250), Left-shift (250) + Left-shift (250) + Mul (450) | 1200 |
| Stage 2 | Add(580) + Left-shift (250), Comparator (300), Add (400) + Add (580) | 1175 |
| Stage 3 | Add(220) + Mul (250) + MUX (250), Mul (600) + MUX (250) | 2126 |
| **Total area** |  | 4501 |
| Case 2 | Stage 1 | Left-shift (465), Left-shift (250) + Mul (600) | 510 |
| Stage 2 | Add (400) + Add (220) + Add (220) + Left-shift (100) + Left-shift (250),  Comparator (300), Add (220) + Add (220) + Mul (450) | 3699 |
| Stage 3 | Mul (600) + MUX (400) | 610 |
| **Total area** |  | 4819 |
| Case 3 | Stage 1 | Add(580) + Left-shift (250), Left-shift (250) + Left-shift (250) + Mul (450) | 1200 |
| Stage 2 | Add(580) + Left-shift (250), Comparator (300), Add (810) | 625 |
| Stage 3 | Add(220) + Mul (250) + MUX (250), Add(220) + Mul (250) + MUX (250) | 3532 |
| **Total area** |  | 5357 |
| Case 4 | Stage 1 | Add(580) + Left-shift (250), Left-shift (250) + Left-shift (250) + Mul (450) | 1200 |
| Stage 2 | Add (400) + Left-shit (250) + Add (220), Comparator (300), Add (400) +  Add (580) | 1831 |
| Stage 3 | Mul (600) + MUX (400) Mul (600) | 970 |
| **Total area** |  | 4001 |

pipeline stage. In the same way, we can find the optimal resource allocation for each pipeline stage. Table 18 shows the optimal resource allocation for case 1 in Figure 29. Here, the total area is 4501. This is how we can find the area for the other cases in Figure 29. Table 19 shows the resource allocation for different pipelining (scheduling) cases of Figure 29. It shows that different pipeline options will require very different types of resource to meet the timing constraint. Table 19 also shows that case 4 results in the smallest area among the four cases. Our combined resource allocation and scheduling algorithm needs to find out the pipelining that results in the lowest area. There can be exponential number of possible pipeline configurations and each pipeline config- uration can have several possible resource allocation strategies. This is why, the problem space of this problem is exponentially large. The problem can be even more complicated if we consider **resource sharing**. Resource sharing allows us to use the same resource to implement multiple operations in the same pipeline stage. Currently, we do not consider resource sharing. To make the problem worse, actual latency and area/power utilization of a pipeline configuration cannot be known until performing actual logic synthesis to generate netlist which itself is a very time consuming process. This is why, existing RTL synthesis tools such as Synopsys Design Compiler generally first performs scheduling with the fastest resource. They then try to improve the area by moving operations between different pipeline stages. Kim et. al. [99] proposed several heuristics to find the optimal resource allocation strategy. However, their solution was primarily developed for data-flow intensive applications, not for control flow intensive applications. Finally, Kondratyev et. al. [103] proposed an approach where they find the most critical operation by evaluating the behavioral timing analysis. Their approach has been integrated with Cadence Stratus high-level synthesis tool. However, area saving with this approach is very minimal. Several researches have

investigated the problem for FPGA [52, 59].

# Future work on broader data-plane function support

Table 20 shows a list of data plane functions and indicates if C2RTL supports that. Future work on C2RTL involves adding support for missing dataplane functions

Table 20: Router data plane functions in C2RTL

|  |  |  |  |
| --- | --- | --- | --- |
| Dataplane Function | Data structure | Candidate data structure | Support |
| Routing table lookup | Trie | CP-Trie [86], Poptrie [7], SAIL [184] | Yes |
| Routing table lookup | Hash | d-left hashing [25], CoLT [135] | No |
| Packet classification | Bit-selecting trie | TabTree [117] | Yes |
| Packet classification | Cutting/splitting tree | CutSplit [116] | No |

# CHAPTER 10

**Conclusion**

To conclude this dissertation, we discuss the broader impacts and outline the areas for future work.

# Broader Impact of CP-Trie

CP-Trie project has several significant broader impacts.

* + - * **Power efficient router.** A router needs to perform several billion lookups per second. Research has shown that almost two thirds of power dissipation inside a core router is due to IP lookup engines [93]. As CP-Trie consumes the power consumption of IP look, it will have significant impact power consumption of the core routers.
      * **Faster IP lookup in Software.** CP-Trie achieves very high throughput while consuming small memory. This makes CP-Trie an attractive solution on software. We have talked to router dataplane developers from Futurewei Technologies and Juniper Networks about CP- Trie. Based on the conversions, CP-Trie lookup throughput seemed very impressive.
      * **ASIC based IP lookup is around** 29× **faster than software based IP lookup.** In the past, it was claimed that software IP lookup, Poptrie can achieve upto 245 Mlps [7]. However, this dissertation points out that such high throughput is achieved when lookup needs to visit only the first level which is very unlikely. In realistic traffic patterns, software IP lookup cannot achieve such a high throughput. In our excrement, CP-Trie achieves around 34 Mlps while Poptrie achieves around 26 Mlps. However, CP-Trie and Poptrie based ASIC can achieve 1 GHz clock speed, thus can perform 1000 Mlps. This is why, ASIC based IP lookup is around 29× faster than software based IP lookup.
      * **CP-Trie for NDN Lookup.** Named Data Networking (NDN) uses longest prefix match on the names [187, 156]. The names in NDN can be significantly longer than IPv6 prefixes. As

CP-Trie can use longer stride, it can be very attractive for NDN lookup compared to other approaches.

* + - * **Open source IP lookup algorithms** We implemented CP-Trie, Poptrie and SAIL based IP lookup algorithms. We made our source code publicly available. This will allow other researchers to improve the solutions. This will also allow open source developer community to include those solutions in those platforms. After discussing with Linux kernel community, we found that they are interested to include CP-Trie, Poptrie and SAIL implementation in Linux kernel.
      * **CP-Trie beyond IP lookup.** Trie (prefix tree) is a widely used data structure for imple- menting dictionary and various search algorithms. CP-Trie describes a new way to encode Trie with bitmaps that is very memory efficient and compute efficient. This is why, CP-Trie can be used in many other applications where regular trie is used. This is how, CP-Trie will have a significant impact beyond IP lookup.

# Broader Impact of C2RTL

* + - * **Bridging the gap between IP lookup and ASIC.** Currently IP lookup in routing ASIC is developed using RTL code which is extremely complex. This is why, only a few companies with huge budgets can afford to develop a routing chip. Such cost refrains academic re- searchers to develop their own chips. As a result, there has been very little research work on routing dataplane algorithms in ASIC. Due to the huge barrier of ASIC, academic researchers often implement their new novel algorithm in software. But, those algorithms almost never make their way into ASIC. This calls for a new routing ASIC development methodology. C2RTL can be a solution to this problem. It allows us to design an algorithm in C and generates corresponding Verilog RTL. This improves productivity significantly. It will allow software developers and networking researchers to develop their novel dataplane algorithm in C and generate corresponding Verilog code. Thus, C2RTL will remove the barrier of routing ASIC development by providing an open source HLS tool for routers. This will improve the hardware and networking research ecosystem significantly. It will also lower the cost of ASIC development by improving the designer productively. C2RTL will also play a critical role in power efficient routing ASIC development.
      * **Open source high-level synthesis for router dataplane.** Currently, most of the open source high-level synthesis tools such as Bambu [133] and LegUp [29] was designed for FPGA, not ASIC, thus they are not suitable for router dataplane. There are several commercial high-level synthesis tools such as Mentor Catapult, Cadence Stratus, NEC CyberWorkBench (CWB), Synopsys Synphony and so on. However, they are not open source, thus not readily accessible for public use. This dissertation presents an open source high-level synthesis tool named C2RTL which will enable router dataplane developers and researchers to design their ASIC at a higher level. Thus, CV2RTL can improve the research ecosystem significantly.
      * **Evaluating IP lookup algorithms in ASIC at high level.** As C2RTL enables us to generate Verilog RTL from IP lookup implementation in C, it will allow researchers to evaluate their new and novel algorithm to implement in C and generate the RTL implementation. They then can evaluate the performance for ASIC using OpenROAD. This is how, C2RTL will enable us to evaluate different network algorithms at a higher level.
      * **Improves designer productivity.** As C2RTL enables us to generate Verilog RTL from IP lookup implementation in C, it will improve designer productivity significantly. Improving designer productivity will reduce the cost of router dataplane development.
      * **An enabler for a power efficient router.** As C2RTL enables us to evaluate different dataplane algorithms and measure their power in ASIC, it will enable us to choose more efficient dataplane algorithms in ASIC.

# Future work on CP-Trie

Besides addressing the limitations described earlier in chapter 9, there are a number of avenues for future work, described below.

* + - * **Implementing CP-Trie on Programmable ASIC.** Recently, programmable ASICs such as Intel/Barefoot Tofino emerged as an alternative to fixed function ASIC. Research should be done on implementing CP-Trie on such architectures.
      * **Dynamic Stride size.** It has been observed that some core routers may have very few prefixes with length 48 − 128. In such cases, it is possible to use much longer stride (e.g. 16-

bit stride) in CP-Trie for those levels whichout increasing memory consumption significantly. A research should be done on implementing CP-Trie with dynamic stride size.

* + - * **CP-Trie for NDN Router.** CP-Trie improves the routing table lookup by using longer strides than Poptrie. CP-Trie can also be useful for Named Data Networking (NDN) lookup. NDN uses longest prefix match on the names [187, 156]. The names in NDN can be sig- nificantly longer than an IPv6 prefix. As CP-Trie uses a longer stride, it would be a more practical solution for NDN prefix match compared to Poptrie. However, NDN prefixes are also variable length. Thus, CP-Trie needs to be modified for NDN lookup. A research should be done on NDN prefix match based on CP-Trie.
      * **Smart memory design for CP-Trie in ASIC.** CP-Trie splits an IPv6 routing table into 15 levels. In pipelined ASIC, each level should be stored in a separate SRAM blocks. Research can be done on optimizing memory design such that multiple levels are stored in the same SRAM block without reducing the throughput. The objective of this approach should be to reduce the number of SRAM blocks needed. Note that, reducing the number of SRAM reduces the area and power consumption significantly. Such memory has been implemented in Broadcom Jericho and Cisco Silicon One ASIC [44, 163, 165].
      * **Routing table lookup in FSM based ASIC.** If ASIC uses finite state machine (FSM) in- stead of pipeline, it will not be able to lookup packets at line rate. However, such architecture has other benefits. It will require just one SRAM module which will improve the power/area significantly. In this case, lookup may stop early if it finds the longest prefix match in a lower level. A research should be done on implementing Trie based IP lookup in FSM based ASIC. Such IP lookup also can be implemented in FPGA.
      * **Novel update mechanism for Trie.** One of the drawbacks of trie based IP lookup is complex update procedure. Here, the update procedure may require updating several levels. Such an update procedure may make the lookup engine temporarily unavailable. Note that, hash based IP lookup consume very large memory and costly lookup, however they allow easier update mechanism. Combining the use of Trie and hash may achieve faster lookup and faster update. Similar mechanisms have been proposed for packet classification. [118] A

research should be done on novel update mechanisms for tries.

* + - * **Multiple lookups simultaneously.** If multiple destination IP addresses to be looked up are very similar, it may be possible to lookup multiple IP addresses while traversing the trie once (with some extra operation). Such lookup scheme will improve the lookup performance significantly.
      * **Comparing Trie based IP lookup with hash based IP lookup.** In this dissertation, we mainly focused on Trie based IP lookup. However, research should be done comparing trie and hash based IP lookup in software and ASIC. Such comparison may reveal the strength and weakness of different approaches.
      * **IP lookup in multiple pipelines.** Routing ASIC generally uses 1 GHz clock speed. Thus, a single core routing ASIC can achieve at most 1*,* 000 million lookups per second. In order, improve the throughput, routing ASIC uses multiple pipelines (similar to multicore architec- ture) [191]. In multiple pipelines, we may need to store multiple copies of the trie. Research should be done on eliminating multiple copies of trie in ASIC.
      * **Evaluate CP-Trie with manufacturable PDK.** Currently, CP-Trie was evaluated with 45nm Nandgate Standard Cell library. The cell library however is not manufacturable. A research should be done on incorporating evaluating CP-Trie with manufacturable PDK such as SkyWater PDK (https://github.com/google/skywater-pdk).
      * **Evaluate CP-Trie with FPGA.** This dissertation primarily focused on IP lookup on soft- ware and ASIC. However, a research should be done on evaluating CP-Trie on FPGA as well. NetFPGA [190] platform can be a suitable platform to do such research.
      * **CP-Trie in Linux XDP, DPDK and VPP.** There are several software dataplane frame- works such as Linux XDP, DPDK and VPP [56, 74]. Those frameworks can be benefited from a better IP lookup algorithm. A research should be done on deploying CP-Trie in Linux XDP, DPDK and VPP.
      * **ASIC tapeout.** A research should be done on integrating the Verilog RTL of CP-Trie with other component of a router. Finally, a research should be done on tapout of the resulting

SoC.

# Future work on C2RTL

Besides addressing the limitations described earlier in chapter 9, there are a number of avenues for future work, described below.

* + - * **Resource allocation algorithm:** Currently, C2RTL mainly focuses on implementing pipelined ASIC. It divides the behavioral description into several pipeline stages where each pipeline stage can be executed in 1ns. It enables us to generate ASIC that can achieve 1 GHz clock speed at line rate. The goal of this project is to develop a resource allocation algorithm that can minimize area and power without violating the timing constraint. An instruction can have multiple resource implementations in a standard cell library. There is a trade-off between delay and area. It has been shown that increasing latency reduces area [99, 103]. It

is noteworthy that reduction in area also reduces power consumption [99]. A research should be done on developing a resource allocation algorithm that reduces area without violating timing constraint. This problem was initially investigated in [104, 99].

* + - * **Broader dataplane function support.** Currently, C2RTL works for trie based IP lookup and packet classification algorithms. We also need to support other dataplane functions, such as, packet parser [64, 11, 120], ethernet lookup [100, 189], OpenFlow packet classification [75], scheduling [150], router-assisted congestion control [82, 83], heavy-hitter detection [149], hash based IP lookup algorithms [135, 134], decomposition based packet classification [58, 57, 138, 137], tree (cutting/splitting) based packet classification [6, 119, 115, 116, 169, 139, 92], bit- selecting trie based packet classification [75, 183, 155], TSS (hash) based packet classification algorithms [118, 43, 132], network monitoring [127], regular expression matching [185, 136], and so on.
      * **Developing benchmark suites for HLS targeting routing chips** Currently, there are two benchmark suites for HLS: CHstone [69] and S2CBench [144]. CHstone [69] benchmark suite contains designs and their testbenches in ANSI-C. ANSI-C is popular when targeting FPGAs as FPGA users are often embedded software engineers who know ANSI-C well. On the other hand, S2CBench [144] is written in SystemC. SystemC is preferred by hardware

designers as it has hardware extensions including data types (fixed and floating point) and allows to model concurrency. However, none of the benchmarks reflects the need for routing chip where applications need to support control flow intensive applications and pipelining. Thus, research should be done on developing a benchmark suit consisting of several dataplane algorithms in order to measure the performance of HLS tools targeting router dataplane.

* + - * **Resource sharing.** Currently, C2RTL does not perform resource sharing [105]. A research should be done on implementing resource sharing in C2RTL.
      * **SystemC support.** Currently, C2RTL only supports C. However, many high-level synthesis applications primarily use SystemC. SystemC is a C++ library for ASIC design. It adds fixed bitwidth, concurrency and few other features needed for hardware design. These features do not exist in standard C++. Research should be done on adopting SystemC in C2RTL. This will allow us to utilize many legacy libraries that are beneficial for developing complete SoC.
      * **Standard Cell library support.** Currently, C2RTL uses 45nm Nandgate Standard Cell library. The cell library however is not manufacturable. A research should be done on incorporating open source SkyWater PDK (https://github.com/google/skywater-pdk). PDK stands for Process Development Kit. They are standard cell library for a semiconductor manufacturing process. The output Verilog of C2RTL should be evaluated with open source PDKs.
      * **Enhanced compiler optimization.** Currently, C2RTL depends on GCC compiler for com- piler optimization. However, GCC often focuses on optimizing the code without sacrificing running time. This is because runtime of a compiler is an important metric. However, in HLS, the output of the compiler is far more important than the runtime of a compiler. It has shown that it is possible to improve QoR of HLS by aggressively optimizing the input code [112]. Such dynamic compiler optimization can be added to C2RTL to enhance its QoR.
      * **Design space exploration.** Automated design space exploration is another benefit of high- level synthesis. Here, the source code of HLS is transformed based on different knobs in order to achieve target objective [145]. A research should be done on implementing design space exploration in C2RTL.
      * **Monolithic 3D IC support.** As Moore’s law is slowing down, semiconductor manufacturing companies are adopting monolithic 3D IC architecture. 29 Currently, all the ICs are 2D. Here, logic and memories are put separately. In monolithic 3D, logic and memories are put on top of each other which will improve performance/power significantly. Monolithic 3D IC is expected to improve the power-performance-area (PPA) of memory intensive applications such as router dataplane applications significantly. A research should be done on using C2RTL for generating Verilog for monolithic 3D IC.
      * **Comparing the QoR with manual RTL code.** The QoRs of HLS-generated RTL code often lag behind those of manual RTL code [166, 109]. A research should be done to compare the QoRs of C2RTL to those of manually written RTL. Currently, C2RTL depends on GCC to optimize the code. However, it has shown that a custom compiler flow can improve the QoRs of HLS [112]. A research should be done on improving QoRs of C2RTL.
      * **Comparing the design productivity with respect to manual RTL design.** A research should be done on measuring design productivity of C2RTL compared to manual RTL design. Such design productivity has been measured in other domains for HLS tools [130].
      * **IP integration and tape out.** The intellectual properties (IP) generated by C2RTL have be integrated with other IPs in a routing SoC. A research should be done on the integration the IPs. We also need to tape-out (sampling the real chip) the resulting SoC to validate our flow.
      * **Power and area estimation.** Currently, C2RTL uses OpenROAD to generate power and area. However, this requires performing physical synthesis process which is not trivial. This may refrain programmers from using C2RTL. A research should be done on estimating power and area without performing actual physical synthesis. This may not be accurate, but it will help developers to get a quick feedback about their implementation. Note that, Alladin [147] performs such simulation. They however mainly focus on data-flow intensive applications. On the other hand, IP lookup and packet classifications are control-flow intensive applications. As a result, the area and power generated by Aladdin for IP and packet classification algorithms

29https://en.wikipedia.org/wiki/Three-dimensional integrated circuit

will have a very high error rate. Research should be done to estimate power and area for control-flow intensive applications.

# Glossary

**API** *Application programming interface*: programmatic interface to interact with a software com- ponent.

**ASIC** *Application specefic integrated circuit* : It is also known as hardware chip or IC. ASIC imple- ments algorithmic solution directly in logic gates. An ASIC is designed for a specific use or application. Contrast this with a CPU: a CPU supports different uses through the software it executes.

**BFD** *Bidirectional Forwarding Detection* : a protocol to detect faults between two routers con- nected by a link.

**BGP** *Border Gateway Protocol* : a protocol for exchanging routing information between routers.

**CMOS** *Complementary metal–oxide–semiconductor* : a type of transistor.

**DRAM** *Dynamic random-access memory or Dynamic RAM* : random-access memory that re- quires periodic refreshing. DRAM has higher capacity and is simpler than SRAM. However, it is slower and requires more power than SRAM. DRAMs are generally used as off-chip memory.

**FIB** *Forwarding information base*: also known as a forwarding table. It contains the prefixes and the corresponding next-hop that determines the forarding behaviour of a router.

**FPGA** *Field programmable gate array* : an integrated circuit in which the logic is configured using a lookup table..

**GDSII** *Graphic design system*: a data format which is the de-facto industry standard for IC layout artwork.

**GPU** *Graphics Processing Unit* : a processor designed for graphics processing. GPUs usually contain many parallel processing pipelines, making them suitable for stream processing—i.e., applying the same operation to a stream of data.

**GRE** *Generic Routing Encapsulation* : a tunneling protocol where an IP packet in encapsulated in another IP packet.

**ICMP** *Internet Control Message Protocol* : a protocol to transmit error and status information between devices on the Internet.

**IGMP** *Internet Group Management Protocol* : a protocol to establish multicast group member- ships.

**IP** *Internet Protocol* : the primary protocol used for communicating data over the Internet. IP encapsulates other protocols such as TCP, UDP, and ICMP.

**IPv4** Version 4 of the Internet Protocol (IP).

**IPv6** Version 6 of the Internet Protocol (IP).

**IS-IS** *Intermediate System to Intermediate System* : a protocol for exchanging routing information between routers.

**LDP** *Label Distribution Protocol* : a protocol for exchanging MPLS label mapping information..

**MPLS** *Multiprotocol Label Switching*: forwarding that uses fixed-length labels instead of IP ad- dresses. Fixed-length label lookups are intended to be simpler than longest-prefix match IP lookups.

**OSPF** *Open Shortest Path First* : a protocol for exchanging routing information between routers.

**SRAM** *Static random-access memory or Static RAM* : random-access memory that does not require periodic refreshing. SRAM is generally faster than DRAM. They are generally used as on-chip memory.

**TCAM** *Ternary content-addressable memory or Ternary CAM* : a content-addressable memory (CAM) that supports “don’t-care” bits that match any value. A CAM lookup searches all memory locations in parallel for the input value. CAMs may be binary or ternary: a binary CAM matches every bit precisely, whereas a ternary CAM (TCAM) allows “don’t-care” bits that match any value. A TCAM can find a match in 1 clock cycle.

**UDP** *User Datagram Protocol* : a transport protocol for unreliable communication between hosts.

**URL** *Uniform resource locator* : a string that specifies a resource on the internet (e.g. [www.](http://www/) google.com).

**VXLAN** *Virtual Extensible LAN* : a layer 2 virtualization protocol to support multi tenancy in cloud data centers.

# BIBLIOGRAPHY

1. T. Ajayi, V. A. Chhabria, M. Fogac¸a, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee,

U. Mallappa, M. Neseem, et al. Toward an open-source digital flow: First learnings from the OpenROAD project. In *ACM/IEEE DAC*, pages 1–4, 2019.

1. G. Antichi, C. Callegari, A. W. Moore, S. Giordano, and E. Anastasi. JA-trie: Entropy-based packet classification. In *IEEE HPSR*, pages 32–37, 2014.
2. Arista 7050X3 Series Switch Architecture. [https://www.arista.com/assets/data/pdf/](http://www.arista.com/assets/data/pdf/) Whitepapers/7050X3\_Architecture\_WP.pdf.
3. Arista 7500R3 Platform Architecture. [https://www.arista.com/assets/data/pdf/](http://www.arista.com/assets/data/pdf/) Whitepapers/Arista7500R3SwitchArchitectureWP.pdf.
4. Arista AlgoMatch. [https://www.arista.com/assets/data/pdf/Whitepapers/AlgoMatch\_](http://www.arista.com/assets/data/pdf/Whitepapers/AlgoMatch_) ProductBrief.pdf.
5. H. Asai. Palmtrie: a ternary key matching algorithm for ip packet filtering rules. In *ACM CoNEXT*, pages 323–335, 2020.
6. H. Asai and Y. Ohara. Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup. In *ACM SIGCOMM 2015*.
7. S. Ataei, M. Gaalswyk, and J. E. Stine. A high performance multi-port SRAM for low voltage shared memory systems in 32 nm CMOS. In *IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1236–1239, 2017.
8. S. Ataei and R. Manohar. AMC: An asynchronous memory compiler. In *IEEE ASYNC*, pages 1–8, 2019.
9. AT&T is First in Industry to Deploy Open Dis-Aggregated Core Router. https://about. att.com/story/2020/open\_disaggregated\_core\_router.html.
10. M. Attig and G. Brebner. 400 gb/s programmable packet parsing on a single fpga. In

*ACM/IEEE ANCS*, pages 12–23, 2011.

1. F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *ACM/IEEE ISCA*, pages 123–133, 2005.
2. J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviˇzienis, J. Wawrzynek, and

K. Asanovi´c. Chisel: constructing hardware in a scala embedded language. In *ACM/IEEE DAC*, pages 1212–1221, 2012.

1. M. Bando, N. S. Artan, and H. J. Chao. FlashLook: 100-gbps hash-tuned route lookup architecture. In *IEEE HPSR*, pages 1–8, 2009.
2. M. Bando and H. J. Chao. FlashTrie: Hash-based prefix-compressed trie for IP route lookup beyond 100Gbps. In *IEEE INFOCOM*, pages 1–9, 2010.
3. A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *IEEE INFOCOM*, pages 64–74, 2003.
4. BGP Report. [http://bgp.potaroo.net/.](http://bgp.potaroo.net/)
5. Bluespec. https://bluespec.com/.
6. P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Ta- layco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet proces- sors. *ACM SIGCOMM CCR*, 44(3):87–95, 2014.
7. P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and

M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM*, 43(4):99–110, 2013.

1. Brodcom Jericho. https://docs.broadcom.com/doc/88690-PB100.
2. Brodcom Tomahawk 3. [https://www.broadcom.com/products/ethernet-connectivity/](http://www.broadcom.com/products/ethernet-connectivity/) switching/strataxgs/bcm56980-series.
3. Brodcom Tomahawk 4. [https://www.broadcom.com/products/ethernet-connectivity/](http://www.broadcom.com/products/ethernet-connectivity/) switching/strataxgs/bcm56990-series.
4. Brodcom Trident. [https://www.broadcom.com/products/ethernet-connectivity/](http://www.broadcom.com/products/ethernet-connectivity/) switching/strataxgs/bcm56880-series.
5. A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP lookups. In

*IEEE INFOCOM*, volume 3, pages 1454–1463, 2001.

1. M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *ACM ASPLOS*, pages 14–26, 2004.
2. Cadence Innovus. [https://www.cadence.com/en\_US/home/tools/](http://www.cadence.com/en_US/home/tools/) digital-design-and-signoff/soc-implementation-and-floorplanning/

innovus-implementation-system.html.

1. Cadence Stratus HLS. [https://www.cadence.com/en\_US/home/tools/](http://www.cadence.com/en_US/home/tools/) digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.
2. A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and

T. Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *ACM FPGA*, pages 33–36, 2011.

1. H. J. Chao and B. Liu. *High performance switches and routers*. John Wiley & Sons, 2007.
2. D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. xPilot: A platform-based behavioral synthesis system. *SRC TechCon*, 5, 2005.
3. A. Ciccardi. *Intelligent Scheduler for Heterogeneous System on Chip*. PhD thesis, 2019.
4. Cisco Enterprise Silicon. [https://www.ciscolive.com/c/dam/r/ciscolive/us/docs/](http://www.ciscolive.com/c/dam/r/ciscolive/us/docs/) 2017/pdf/BRKARC-3467.pdf.
5. Cisco NCS 5500 Modular Platform Architecture. [https://www.cisco.com/c/dam/](http://www.cisco.com/c/dam/) en/us/products/collateral/routers/network-convergence-system-5500-series/ ncs5500-modular-platform-architecture-white-paper.pdf.
6. Cisco Nexus 3000 Switch Architecture. [https://www.ciscolive.com/c/dam/r/ciscolive/](http://www.ciscolive.com/c/dam/r/ciscolive/) us/docs/2018/pdf/BRKDCN-3734.pdf.
7. Cisco Silicon One. [https://www.cisco.com/c/en/us/solutions/collateral/](http://www.cisco.com/c/en/us/solutions/collateral/) silicon-one/datasheet-c78-744312.html.
8. Cisco Silicon One: ONE Silicon, ONE Experience, MULTIPLE Roles. https://blogs. cisco.com/sp/one-silicon-one-experience-multiple-roles.
9. N. I. C. Compiler. *Nios II C2H Compiler*. Altera Corporation (Intel), 2010.
10. J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, 2011.
11. P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin. GAUT: A high-level synthesis tool for DSP applications. In *High-Level Synthesis*, pages 147–169. Springer, 2008.
12. P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis.

*IEEE Design & Test of Computers*, 26(4):8–17, 2009.

1. Cumulus Network. https://cumulusnetworks.com/.
2. J. Daly, V. Bruschi, L. Linguaglossa, S. Pontarelli, D. Rossi, J. Tollet, E. Torng, and

A. Yourtchenko. Tuplemerge: Fast software packet processing for online packet classifica- tion. *IEEE/ACM transactions on networking*, 27(4):1417–1431, 2019.

1. S. Das. Smart-Table Technology - Enabling Very Large Server, Storage Nodes and Virtual Machines to Scale Using Flexible Network Infrastructure Topologies. *BROADCOM Corpo- ration StrataXGS White Paper*, 2012.
2. L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. PLUG: flexible lookup modules for rapid deployment of new protocols in high-speed routers. In *ACM SIGCOMM*, pages 207–218, 2009.
3. G. De Micheli. *Synthesis and optimization of digital circuits*. Number BOOK. McGraw Hill, 1994.
4. M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *ACM SIGCOMM*, 27(4), 1997.
5. Dolphin Technology’s Memory compiler. [http://www.dolphin-ic.com/memory-products.](http://www.dolphin-ic.com/memory-products) html.
6. DriveNets is Building Networks Like Cloud. [https://www.youtube.com/watch?v=](http://www.youtube.com/watch?v) HMNa8XYnfsI&t=102s.
7. W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.
8. J. P. Elliott. *Understanding behavioral synthesis: a practical guide to high-level design*. Springer Science & Business Media, 1999.
9. F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 29(6):911–924, 2010.
10. M. Fingeroff. *High-level synthesis: blue book*. Mentor Graphics Corporation, 2010.
11. J. Fu, O. Hagsand, and G. Karlsson. Performance evaluation and cache behavior of LC-trie for IP-address lookup. In *IEEE HPSR*, pages 7–pp, 2006.
12. D. D. Gajski, J. Zhu, R. D¨omer, A. Gerstlauer, and S. Zhao. *SpecC: Specification language and methodology*. Springer Science & Business Media, 2012.
13. S. Gallenmu¨ller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of frame- works for high-performance packet IO. In *ACM/IEEE ANCS*, 2015.
14. T. Ganegedara, W. Jiang, and V. K. Prasanna. A scalable and modular architecture for high- performance packet classification. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1135–1144, 2013.
15. T. Ganegedara and V. K. Prasanna. StrideBV: Single chip 400G+ packet classification. In

*IEEE HPSR*, pages 1–6, 2012.

1. I. Ganusov, H. Fraisse, A. Ng, R. T. Possignolo, and S. Das. Automated extra pipeline analysis of applications mapped to xilinx ultrascale+ fpgas. In *IEEE FPL*, pages 1–10, 2016.
2. J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and

M. Yu. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*, pages 435–450, 2020.

1. X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman,

S. Narayana, and A. Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, pages 44–61, 2020.

1. GCC Plugin. https://gcc.gnu.org/wiki/GCC\_Plugins.
2. A. A. Ghazy and M. Shalan. OpenLANE: The open-source digital ASIC implementation flow. In *WOSET*, 2020.
3. G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *ACM/IEEE ANCS*, pages 13–24, 2013.
4. T. Gr¨otker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer Science & Business Media, 2007.
5. S. Gupta, R. Gupta, N. D. Dutt, and A. Nicolau. *SPARK: a parallelizing approach to the high-level synthesis of digital circuits*. Springer Science & Business Media, 2007.
6. M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar. OpenRAM: An open-source memory compiler. In *IEEE/ACM ICCAD*, pages 1–6, 2016.
7. D. Hanks and H. Reynolds. *Juniper MX series*. ” O’Reilly Media, Inc.”, 2012.
8. Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *IEEE ISCAS*, pages 1192–1195, 2008.
9. E. N. Harris, S. L. Wasmundt, L. De Carli, K. Sankaralingam, and C. Estan. LEAP: Latency- energy-and area-optimized lookup pipeline. In *ACM/IEEE ANCS*, pages 175–186, 2012.
10. J. Hasan, S. Cadambi, V. Jakkula, and S. Chakradhar. Chisel: A storage-efficient, collision- free hash-based network processing architecture. In *ACM/IEEE ISCA*, pages 203–215, 2006.
11. J. Hasan and T. Vijaykumar. Dynamic pipelining: making IP-lookup truly scalable. In *ACM SIGCOMM*, volume 35, pages 205–216, 2005.
12. Y. Herklotz, J. D. POLLARD, N. Ramanathan, and J. Wickerson. Formal verification of high-level synthesis. *ACM OOPSLA*, 2021.
13. T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and

D. Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *ACM CoNEXT*, 2018.

1. C.-L. Hsieh and N. Weng. Many-field packet classification for software-defined networking switches. In *ACM/IEEE ANCS*, pages 13–24, 2016.
2. K. Huang, G. Xie, Y. Li, and A. X. Liu. Offset addressing approach to memory-efficient IP address lookup. In *IEEE INFOCOM*, pages 306–310, 2011.
3. C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 10(4):464–475, 1991.
4. Icarus Verilog. [http://iverilog.icarus.com/.](http://iverilog.icarus.com/)
5. IEEE 1800-2005 - IEEE Standard for SystemVerilog. https://standards.ieee.org/ standard/1800-2005.html.
6. Intel HLS Compiler. [https://www.intel.co.uk/content/www/uk/en/software/](http://www.intel.co.uk/content/www/uk/en/software/) programmable/quartus-prime/hls-compiler.html.
7. Intel OpenCL SDK. https://software.intel.com/content/www/us/en/develop/tools/ opencl-sdk.html.
8. M. I. Islam and J. I. Khan. A network-centric TCP for interactive video delivery networks (VDN). In *IEEE ICNP*, pages 1–6, 2017.
9. M. I. Islam and J. I. Khan. Leveraging Domino to implement RCP in a stateful programmable pipeline. In *IEEE HPSR*, 2019.
10. M. I. Islam and J. I. Khan. SAIL based FIB lookup in a programmable pipeline based Linux router. In *IEEE HPSR*, pages 1–8, 2019.
11. M. I. Islam and J. I. Khan. C2RTL: A high-level synthesis system for IP lookup and packet classification. In *IEEE HPSR*, 2021.
12. M. I. Islam and J. I. Khan. CP-Trie: Cumulative PopCount based Trie for IPv6 Routing Table Lookup in Software and ASIC. In *IEEE HPSR*, 2021.
13. G. Jacobson. Space-efficient static trees and graphs. In *IEEE Annual Symposium on Foun- dations of Computer Science*, pages 549–554, 1989.
14. H. Jamil and N. Weng. Multibit Tries Packet Classification with Deep Reinforcement Learn- ing. In *IEEE HPSR*, pages 1–6, 2020.
15. W. Jiang and V. K. Prasanna. Parallel ip lookup using multiple sram-based pipelines. In

*IEEE IPDPS*, pages 1–14, 2008.

1. W. Jiang and V. K. Prasanna. Large-scale wire-speed packet classification on FPGAs. In

*ACM FPGA*, pages 219–228, 2009.

1. W. Jiang and V. K. Prasanna. Architecture-aware data structure optimization for green ip lookup. In *IEEE HPSR*, pages 113–118, 2010.
2. W. Jiang and V. K. Prasanna. Scalable packet classification on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(9):1668–1680, 2011.
3. W. Jiang and V. K. Prasanna. Data structure optimization for power-efficient ip lookup architectures. *IEEE Transactions on Computers*, 62(11):2169–2182, 2012.
4. W. Jiang, Q. Wang, and V. K. Prasanna. Beyond TCAMs: An SRAM-based parallel multi- pipeline architecture for terabit IP lookup. In *IEEE INFOCOM*, pages 1786–1794, 2008.
5. Juniper Express. https://blogs.juniper.net/en-us/

service-provider-transformation/choosing-a-silicon-architecture-its-all-about-finding-the

1. A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu. *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media, 2011.
2. B. Khailany, E. Krimer, R. Venkatesan, J. Clemons, J. S. Emer, M. Fojtik, A. Klinefelter,

M. Pellauer, N. Pinckney, Y. S. Shao, et al. A modular digital VLSI flow for high-productivity SoC design. In *ACM/IEEE DAC*, pages 1–6, 2018.

1. K. S. Khouri, G. Lakshminarayana, and N. K. Jha. IMPACT: A high-level synthesis system for low power control-flow intensive circuits. In *IEEE DATE*, pages 848–854, 1998.
2. L.-W. Kim, D.-U. Lee, and J. Villasenor. Automated iterative pipelining for ASIC design. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(2):1–24, 2015.
3. S. J. Kim, L. De Carli, K. Sankaralingam, and C. Estan. SWSL: software synthesis for network lookup. In *ACM/IEEE ANCS*, pages 191–202, 2013.
4. A. Kirsch and M. Mitzenmacher. The power of one move: Hashing schemes for hardware.

*IEEE/ACM Transactions on Networking*, 18(6):1752–1765, 2010.

1. D. W. Knapp. *Behavioral synthesis: digital system design using the synopsys behavioral compiler*. Prentice-Hall, Inc., 1996.
2. A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Realistic performance-constrained pipelining in high-level synthesis. In *IEEE DATE*, pages 1–6, 2011.
3. A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Exploiting area/delay tradeoffs in high-level synthesis. In *IEEE DATE*, pages 1024–1029, 2012.
4. A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Share with care: A quantitative evaluation of sharing approaches in high-level synthesis. In *IEEE DATE*, pages 1547–1552, 2013.
5. D. Ku and G. DeMicheli. *HardwareC–a language for hardware design (version 2.0)*. Stanford University, 1990.
6. S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: fast and efficient IP lookup archi- tecture. In *ACM/IEEE ANCS*, pages 51–60, 2006.
7. S. Kundu, S. Lerner, and R. Gupta. Validating high-level synthesis. In *International Con- ference on Computer Aided Verification*, pages 459–472. Springer, 2008.
8. S. Lahti, P. Sj¨ovall, J. Vanne, and T. D. H¨am¨al¨ainen. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(5):898–911, 2018.
9. B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software- defined networks. In *ACM SIGCOMM HotNet*, 2010.
10. M. Lattuada and F. Ferrandi. Code transformations based on speculative SDC scheduling. In *IEEE/ACM ICCAD*, pages 71–77, 2015.
11. M. Lattuada and F. Ferrandi. A design flow engine for the support of customized dynamic high level synthesis flows. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(4):1–26, 2019.
12. C.-H. Lee, C.-H. Shih, J.-D. Huang, and J.-Y. Jou. Equivalence checking of scheduling with speculative code transformations in high-level synthesis. In *IEEE ASP-DAC*, pages 497–502, 2011.
13. S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture- level modeling for SRAM-based structures with advanced leakage reduction techniques. In *ACM/IEEE ICCAD*, pages 694–701, 2011.
14. W. Li and X. Li. HybridCuts: A scheme combining decomposition and cutting for packet classification. In *IEEE High-Performance Interconnects*, pages 41–48, 2013.
15. W. Li, X. Li, H. Li, and G. Xie. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM*, pages 2645–2653, 2018.
16. W. Li, T. Yang, Y.-K. Chang, T. Li, and H. Li. TabTree: A TSS-assisted bit-selecting tree scheme for packet classification with balanced rule mapping. In *ACM/IEEE ANCS*, pages 1–8, 2019.
17. W. Li, T. Yang, O. Rottenstreich, X. Li, G. Xie, H. Li, B. Vamanan, D. Li, and H. Lin. Tuple space assisted packet classification with high performance on both search and update. *IEEE Journal on Selected Areas in Communications (JSAC)*, 38(7):1555–1569, 2020.
18. E. Liang, H. Zhu, X. Jin, and I. Stoica. Neural packet classification. In *ACM SIGCOMM*, pages 256–269. 2019.
19. H. Liu, Z. Qiu, W. Pan, J. Li, and J. Huang. HyperParser: A High-Performance Parser Architecture for Next Generation Programmable Switch and SmartNIC. *ACM APNet*, 2021.
20. Y. Liu, B. Zhang, and L. Wang. Fifa: Fast incremental fib aggregation. In *IEEE INFOCOM*, pages 1–9, 2013.
21. D. Lockhart, G. Zibrat, and C. Batten. PyMTL: A unified framework for vertically integrated computer architecture research. In *IEEE/ACM Micro*, pages 280–292, 2014.
22. Mellanox Spectrum. [https://www.mellanox.com/files/doc-](http://www.mellanox.com/files/doc-2020/pb-spectrum-3.pdf)2020/pb-[spectrum-3.pdf.](http://www.mellanox.com/files/doc-2020/pb-spectrum-3.pdf)
23. O. Mencer. ASC: a stream compiler for computing with FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(9):1603–1617, 2006.
24. R. Nane, V.-M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. Dwarv 2.0: A cosy-based c-to-vhdl hardware compiler. In *IEEE FPL*, pages 619–622, 2012.
25. R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown,

F. Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transac- tions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35(10):1591– 1604, 2015.

1. S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and

C. Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, pages 85–98, 2017.

1. H. Nichols, M. Grimes, J. Sowash, J. Cirimelli-Low, and M. R. Guthaus. Automated Synthesis of Multi-Port Memories and Control. In *IFIP/IEEE VLSI-SoC*, pages 59–64, 2019.
2. S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE Journal on selected Areas in Communications (JSAC)*, 17(6):1083–1092, 1999.
3. M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry. Design productivity of a high level synthesis compiler versus hdl. In *IEEE SAMOS*, pages 140–147, 2016.
4. D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad,

S. Canakci, B. Veluri, et al. Blackparrot: An agile open-source risc-v multicore for accel- erator socs. *IEEE Micro*, 40(4):93–102, 2020.

1. B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang,

J. Stringer, P. Shelar, et al. The design and implementation of open vswitch. In *USENIX NSDI*, pages 117–130, 2015.

1. C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *IEEE FPL*, pages 1–4, 2013.
2. F. Pong and N.-F. Tzeng. SUSE: Superior storage-efficiency for routing tables through prefix transformation and aggregation. *IEEE/ACM Transactions on Networking*, 18(1):81–94, 2009.
3. F. Pong and N.-F. Tzeng. Concise lookup tables for IPv4 and IPv6 longest prefix matching in scalable routers. *IEEE/ACM Transactions on Networking*, 20(3):729–741, 2011.
4. Y. Qu, Y.-H. E. Yang, and V. K. Prasanna. Large-scale multi-flow regular expression matching on fpga. In *IEEE HPSR*, pages 70–75, 2012.
5. Y. R. Qu and V. K. Prasanna. High-performance and dynamically updatable packet classifica- tion engine on fpga. *IEEE Transactions on Parallel and Distributed Systems*, 27(1):197–209, 2015.
6. Y. R. Qu, S. Zhou, and V. K. Prasanna. High-performance architecture for dynamically updatable packet classification on fpga. In *ACM/IEEE ANCS*, pages 125–136, 2013.
7. A. Rashelbach, O. Rottenstreich, and M. Silberstein. A computational approach to packet classification. In *ACM SIGCOMM*, pages 542–556, 2020.
8. G. R´etv´ari, J. Tapolcai, A. K˝or¨osi, A. Majd´an, and Z. Heszberger. Compressing IP forwarding tables: towards entropy bounds and beyond. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 111–122, 2013.
9. R. Rojas-Cessa, T. Kijkanjanarat, W. Wangchai, K. Patil, and N. Thirapittayatakul. Helix: IP lookup scheme based on helicoidal properties of binary trees. *Computer Networks*, 89:78– 89, 2015.
10. Route Views Project. [http://www.routeviews.org/routeviews/.](http://www.routeviews.org/routeviews/)
11. J. Sanguinetti, M. Meredith, and S. Dart. Transaction-accurate interface scheduling in high- level synthesis. In *2012 Electronic System Level Synthesis Conference*, pages 31–36. IEEE, 2012.
12. B. C. Schafer and A. Mahapatra. S2cbench: Synthesizable systemc benchmark suite for high-level synthesis. *IEEE Embedded Systems Letters*, 6(3):53–56, 2014.
13. B. C. Schafer and Z. Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(10):2628–2639, 2019.
14. R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High- level synthesis of nonprogrammable hardware accelerators. In *IEEE ASAP*, pages 113–124, 2000.
15. Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ACM/IEEE ISCA*, volume 42, pages 97–108, 2014.
16. S. Singh and D. J. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *IEEE FCCM*, pages 3–12, 2008.
17. A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese,

N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*, pages 15–28, 2016.

1. A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh,

S. Katti, N. McKeown, and H. Balakrishnan. Towards programmable packet scheduling. In

*ACM SIGCOMM*, pages 1–7, 2015.

1. A. D. Solutions. Handel-c language reference manual. agility design solutions, 2007.
2. H. Song, F. Hao, M. Kodialam, and T. Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100Gbps core router line cards. In *IEEE INFOCOM*, pages 2518– 2526, 2009.
3. H. Song, M. Kodialam, F. Hao, and T. Lakshman. Scalable IP lookups using shape graphs. In *IEEE ICNP*, pages 73–82, 2009.
4. H. Song, J. Turner, and J. Lockwood. Shape shifting tries for faster IP route lookup. In

*IEEE ICNP*, pages 10–pp, 2005.

1. H. Song and J. S. Turner. ABC: Adaptive binary cuttings for multidimensional packet clas- sification. *IEEE/ACM Transactions On Networking*, 21(1):98–109, 2012.
2. T. Song, H. Yuan, P. Crowley, and B. Zhang. Scalable name-based packet forwarding: From millions to billions. In *ACM ICN*, pages 19–28, 2015.
3. V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. In *ACM SIGMETRICS*, volume 26, pages 1–10, 1998.
4. Standard cell library. https://en.wikipedia.org/wiki/Standard\_cell.
5. Static Single Assignment (SSA) form. https://en.wikipedia.org/wiki/Static\_single\_ assignment\_form.
6. Static Single Assignment (SSA) in GCC. https://gcc.gnu.org/onlinedocs/gcc-4.3.4/ gccint/SSA.html.
7. Synopsys Fusion Compiler. [https://www.synopsys.com/implementation-](http://www.synopsys.com/implementation-and-signoff/)and-[signoff/](http://www.synopsys.com/implementation-and-signoff/) physical-implementation/fusion-compiler.html.
8. Tech Field Day talk: Barefoot Tofino. [https://www.youtube.com/watch?v=BqOp6w\_7cGI](http://www.youtube.com/watch?v=BqOp6w_7cGI).
9. Tech Field Day talk: Broadcom Jericho2 - Driving the Merchant Silicon Revolution in Service Provider Networks. [https://www.youtube.com/watch?v=n9bUEb1v0pU](http://www.youtube.com/watch?v=n9bUEb1v0pU).
10. Tech Field Day talk: Broadcom Tomahawk4. [https://www.youtube.com/watch?v=](http://www.youtube.com/watch?v) B-COGMbaUg4.
11. Tech Field Day talk: Cisco Silicon One. [https://www.youtube.com/watch?v=KIGct1QOtdI](http://www.youtube.com/watch?v=KIGct1QOtdI).
12. The evolution of high-level synthesis. https://semiengineering.com/ the-evolution-of-high-level-synthesis/.
13. B. Tian, J. Gao, M. Liu, E. Zhai, Y. Chen, Y. Zhou, L. Dai, F. Yan, M. Ma, M. Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *ACM SIGCOMM 2021*, pages 17–32.
14. Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/.
15. D. Tong, Y. R. Qu, and V. K. Prasanna. High-throughput traffic classification on multi-core processors. In *IEEE HPSR*, pages 138–145, 2014.
16. J. L. Tripp, K. D. Peterson, C. Ahrens, J. D. Poznanovic, and M. B. Gokhale. Trident: an FPGA compiler framework for floating-point algorithms. In *IEEE FPL, 2005.*, pages 317–322, 2005.
17. R. S. Tucker, R. Parthiban, J. Baliga, K. Hinton, R. W. Ayre, and W. V. Sorin. Evolution of wdm optical ip networks: A cost and energy perspective. *IEEE Journal of Lightwave Technology*, 27(3):243–252, 2009.
18. Z. A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis. SMALTA: practical and near-optimal FIB aggregation. In *ACM CoNEXT*, pages 1–12, 2011.
19. N. Vaish, T. Kooburat, L. De Carli, K. Sankaralingam, and C. Estan. Experiences in co- designing a packet classification algorithm and a flexible hardware platform. In *ACM/IEEE ANCS*, pages 189–199, 2011.
20. J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *IEEE FCCM*, pages 127–134, 2010.
21. K. Wakabayashi and B. C. Schafer. All-in-C: Behavioral Synthesis and Verification with CyberWorkBench. In *High-Level Synthesis*, pages 113–127. Springer, 2008.
22. H. S. Warren. *Hacker’s delight*. Pearson Education, 2013.
23. B. Wheeler. Tomahawk 4 switch first to 25.6 tbps. *Microprocessor Report-Newsletter*, 2019.
24. WIDE Project MAWI Working Group Traffic Archive. [http://mawi.wide.ad.jp/mawi/.](http://mawi.wide.ad.jp/mawi/)
25. x86 Instruction Set Reference. https://c9x.me/x86/.
26. Xilinx SDAccel. [https://www.xilinx.com/products/design-](http://www.xilinx.com/products/design-tools/software-zone/)tools/s[oftware-zone/](http://www.xilinx.com/products/design-tools/software-zone/) sdaccel.html.
27. Cavium XPliant. [http://www.cavium.com/pdfFiles/CNX880XX\_PB\_Rev1.pdf.](http://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf)
28. Xsight Lab X1. https://xsightlabs.com/x1-ethernet-switches/.
29. B. Yang, J. Fong, W. Jiang, Y. Xue, and J. Li. Practical multituple packet classification using dynamic discrete bit selection. *IEEE Transactions on Computers*, 63(2):424–434, 2012.
30. T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee IP lookup performance with FIB explosion. In *ACM SIGCOMM 2014*.
31. Y.-H. E. Yang, W. Jiang, and V. K. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *ACM/IEEE ANCS*, pages 30–39, 2008.
32. M. Zec, L. Rizzo, and M. Mikuc. DXR: towards a billion routing lookups per second in software. *ACM SIGCOMM Computer Communication Review*, 42(5):29–36, 2012.
33. L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos,

L. Wang, and B. Zhang. Named data networking. *ACM SIGCOMM Computer Communica- tion Review*, 44(3):66–73, 2014.

1. H. Zheng, S. T. Gurumani, L. Yang, D. Chen, and K. Rupnow. High-level synthesis with behavioral-level multicycle path analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 33(12):1832–1845, 2014.
2. D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with CuckooSwitch. In *ACM CoNEXT*, pages 97–108, 2013.
3. N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE micro*, 34(5):32–41, 2014.
4. N. Zilberman, G. Bracha, and G. Schzukin. Stardust: Divide and conquer in the data center network. In *USENIX NSDI*, pages 141–160, 2019.