

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Range Trie Software Support for Longest Prefix Match

Charalampos Mastrogeorgopoulos

CE-MS-2011-28

Abstract

The range trie constitutes an advanced address lookup scheme aiming at low lookup throughput, latency and memory requirements. The range trie structure has been enhanced with longest prefix match and updating capabilities in order to simulate the forwarding process performed in network routers. For the main work involved in this thesis we fixed, improved, enhanced the performance and introduced additional functionality to the software implementation of the heuristics which generate high quality range trie configurations based on the bounds extracted from the lookup tables. With our work, the software is capable of producing high quality configurations for a great range of possible range trie hardware implementations, for any address length (IPv4 and IPv6 among others), and for any size of lookup tables in relatively low time. As supplementary work we developed the dynamic support software for an IPv4 range trie structure hardware implementation on the HTX reconfigurable platform. The software translates the generated configuration into suitable commands for the software/hardware interface and also offers all the necessary functionality of verifying and maintaining the hardware range trie configuration.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Range Trie Software Support for Longest Prefix Match

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Charalampos Mastrogeorgopoulos
born in Thessaloniki, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Range Trie Software Support for Longest Prefix Match

by Charalampos Mastrogeorgopoulos

Abstract

The range trie constitutes an advanced address lookup scheme aiming at low lookup throughput, latency and memory requirements. The range trie structure has been enhanced with longest prefix match and updating capabilities in order to simulate the forwarding process performed in network routers. For the main work involved in this thesis we fixed, improved, enhanced the performance and introduced additional functionality to the software implementation of the heuristics which generate high quality range trie configurations based on the bounds extracted from the lookup tables. With our work, the software is capable of producing high quality configurations for a great range of possible range trie hardware implementations, for any address length (IPv4 and IPv6 among others), and for any size of lookup tables in relatively low time. As supplementary work we developed the support software for an IPv4 range trie structure hardware implementation on the HTX reconfigurable platform. The software translates the generated configuration into suitable commands for the software/hardware interface and also offers all the necessary functionality of verifying and maintaining the hardware range trie configuration.

Laboratory : Computer Engineering
Codenumber : CE-MS-2011-28

Committee Members :

Advisor: Ioannis Sourdis, CSE, Chalmers Univ. of Tech.

Member: Koen Bertels, CE, TU Delft

Member: Georgi Gaydadjiev, CE, TU Delft

Member: Christian Doerr, NAS, TU Delft

This thesis is dedicated to all those who supported and inspired me, close and far…

Contents

List of Figures vii

List of Tables ix

List of Acronyms xi

Acknowledgements xiii

1. Introduction 1
- 1.1. Address Lookup 4
- 1.2. The Problem 6
- 1.3. Thesis Objectives/Work 7
- 1.4. Thesis Overview 7

2. Background 9
- 2.1. Address Lookup Related Algorithms 11
  - 2.1.1. Binary Trie 12
  - 2.1.2. Path-Compressed Trie 13
  - 2.1.3. Multi-bit Trie 15
  - 2.1.4. Level-Compressed Tries 16
  - 2.1.5. Tree Bitmap 17
  - 2.1.6. Binary Search on Length 18
  - 2.1.7. Linear Search 19
  - 2.1.8. Range Tree 20
  - 2.1.9. Bloom Filters 21
  - 2.1.10. Hardware Oriented Solutions 22
- 2.2. Range Trie 22
  - 2.2.1. Range Trie Concepts and Rules 24
    - 2.2.1.1. Rule 1 – Omit the common prefix of the node 25
    - 2.2.1.2. Rule 2 – Share addresses' common prefix 26
    - 2.2.1.3. Rule 3 – Share addresses' common suffix 27
    - 2.2.1.4. Rule 4 – Omit addresses zeroes suffix 29
    - 2.2.1.5. Rule 5 – Address alignment 29
  - 2.2.2. Range trie Structure 31
  - 2.2.3. Range Trie Construction 32
  - 2.2.4. Range Trie Support for Longest Prefix Match 34
  - 2.2.5. Dynamic Range Tries 35
- 2.3. Summary 36

3. Range Trie Configuration Generation Software **39**
3.1. Software Modifications and Additions 40
3.1.1. Software Correctness and Reliability 42
3.1.2. Software Flexibility/Versatility 45
3.1.3. Software Performance 49
3.1.3.1. Memory Management 49
3.1.3.2. Execution Time Reduction 52
3.1.4. Range Tries Quality 53
3.2. Experimental Results 53
3.2.1. Highest Quality Range Tries 54
3.2.2. Comparison of VLC – V/SLC – SLC Range Tries 61
3.2.3. Rules' Effectiveness Measurements 67
3.2.4. Results for Simplified Range Trie 71
3.3. Conclusions 76

4. HTX Platform Range Trie Implementation Support Software **77**
4.1. The HTX Platform 77
4.2. Software/Hardware Communication 79
4.3. Range Trie Initialization 81
4.4. Range Trie Maintenance/Verification 82
4.4.1. Prefixes Theory 83
4.4.2. Reading the Fixed-levels Part 84
4.4.3. Reading the Spare-levels Part 88
4.4.4. Balancing the Spare Levels 90
4.4.5. Total Range Trie Restructuring 94
4.5. Summary 95

5. Conclusions **97**
5.1 Summary 97
5.2 Contributions 98
5.3 Suggestions/Future Work 100

Bibliography **101**

A. Example Configuration File **103**

B. Software Testing Results **107**

C. Example "makefile" for the HTX Platform Support Software **111**

D. HTX Software Commands – Hardware Output **113**

List of Figures

Figure 1.1: Internet Users Growth 1
Figure 1.2: Internet Traffic Growth 2
Figure 1.3: Internet Host Count Growth 2
Figure 1.4: Example Network Topology 3
Figure 1.5: Network Router 3
Figure 1.6: IPv4 Address 5
Figure 1.7: IPv4 Routing Table 5
Figure 1.8: IPv4 Address Lookup 5

Figure 2.1: Classful Addressing 10
Figure 2.2: CIDR Example 10
Figure 2.3: Address Lookup Approaches 12
Figure 2.4: Binary Trie 13
Figure 2.5: Path-Compressed Trie 14
Figure 2.6: Multi-bit Trie 15
Figure 2.7: Level-Compressed Trie 16
Figure 2.8: Tree Bitmap 17
Figure 2.9: Binary Search on Length 19
Figure 2.10: Binary Range Tree 20
Figure 2.11: 4-ways Range Tree 21
Figure 2.12: Bloom Filters 22
Figure 2.13: Range Trie Node 24
Figure 2.14: Bounds Passing/Child Node Sharing 24
Figure 2.15: Rule 1 – Omit Common Prefix 26
Figure 2.16: Rule 2 – Share Common Prefix 27
Figure 2.17: Rule 3 – Share Common Suffix 28
Figure 2.18: Rule 4 – Omit Zero Suffix 29
Figure 2.19: Rule 5 – Address Alignment 30
Figure 2.20: Dynamic Range Trie 36

Figure 3.1: 128-bits comparator splitting, parcing, encoding 47
Figure 3.2: Reordered Processing Sequence 50
Figure 3.3: Node Processing Scheduling – Control and Data Flow 51
Figure 3.4: VLC: Number of Levels vs. Number of Prefixes 55
Figure 3.5: VLC: Memory in bytes vs. Number of Prefixes 55
Figure 3.6: VLC: Number of Bounds vs. Number of Prefixes 56
Figure 3.7: VLC: Execution Time vs. Number of Prefixes 57
Figure 3.8: VLC IPv4/256 bits MEMBW: Nodes per Level 58
Figure 3.9: VLC IPv6/256 bits MEMBW: Nodes per Level 58
Figure 3.10: VLC IPv6/1024 bits MEMBW: Nodes per Level 59

Figure 3.11: VLC IPv4/256 bits MEMBW: Average Branching Factor per Level 59
Figure 3.12: VLC IPv6/256 bits MEMBW: Average Branching Factor per Level 60
Figure 3.13: VLC IPv6/1024 bits MEMBW: Average Branching Factor per Level 60
Figure 3.14: IPv4: VLC – V/SLC – SLC: Number of Levels vs. Number of Prefixes 62
Figure 3.15: IPv6: VLC – V/SLC – SLC: Number of Levels vs. Number of Prefixes 62
Figure 3.16: IPv4: V/SLC – SLC: Percentage Memory Increase vs. Number of Prefixes 63
Figure 3.17: IPv6: V/SLC – SLC: Percentage Memory Increase vs. Number of Prefixes 63
Figure 3.18: IPv4: VLC – V/SLC – SLC: Number of Bounds per Node vs. Number of Prefixes 64
Figure 3.19: IPv6: VLC – V/SLC – SLC: Number of Bounds per Node vs. Number of Prefixes 65
Figure 3.20: IPv4: VLC – V/SLC – SLC: Execution Time vs. Number of Prefixes 66
Figure 3.21: IPv6: VLC – V/SLC – SLC: Execution Time vs. Number of Prefixes 66
Figure 3.22: Simplified RT, VLC/IPv4/256 bits Memory Bandwidth: Number of Levels vs. Number of Prefixes 72
Figure 3.23: Simplified RT, VLC/IPv6/256 bits Memory Bandwidth: Number of Levels vs. Number of Prefixes 73
Figure 3.24: Simplified RT, VLC/IPv4/256 bits Memory Bandwidth: Memory vs. Number of Prefixes 74
Figure 3.25: Simplified RT, VLC/IPv6/256 bits Memory Bandwidth: Memory vs. Number of Prefixes 74
Figure 3.26: Simplified RT, VLC/IPv4/256 bits Memory Bandwidth: Number of Bounds per Node vs. Number of Prefixes 75
Figure 3.27: Simplified RT, VLC/IPv6/256 bits Memory Bandwidth: Number of Bounds per Node vs. Number of Prefixes 75

Figure 4.1: HTX Platform 78
Figure 4.2: Wrapper scheme based on queues 78
Figure 4.3: Original Bounds Extraction Process 87
Figure 4.4: Spare part Pointer Translation 89
Figure 4.5: Imbalanced Spare part Sub-tree 90
Figure 4.6: Rebalanced Spare part Sub-tree 92
Figure 4.7: Balanced Sub-tree: Number of Levels vs. Number of Bounds 93
Figure 4.8: Balanced Sub-tree: Number of Nodes vs. Number of Bounds 93
Figure 4.9: Balanced Sub-tree: Worst Case Bounds Tolerance vs. Number of Bounds 94

Figure D.1: Communication Commands – Format 114

List of Tables

Table 2.1: Example Lookup Table 11

Table 3.1: 32 bits Comparators – Validation Process for 4 Subsequent Comparators 48
Table 3.2: IPv4: Number of Nodes with specific rules Properties 68
Table 3.3: IPv6: Number of Nodes with specific rules Properties 68
Table 3.4: IPv4: Number of Nodes per Level with different applied rules 70
Table 3.5: IPv6: Number of Nodes per Level with different applied rules 70
Table 3.6: IPv4: Number of Nodes per Level with different applied rules and added Full Bandwidth Utilization 71
Table 3.7: IPv6: Number of Nodes per Level with different applied rules and added Full Bandwidth Utilization 71

Table 4.1: Prefixes/Lookup Table Example 83
Table 4.2: 32-bits comparisons splitting 84

Table B.1: VLC IPv4/256 bits MEMBW: Number of Nodes per Level 107
Table B.2: VLC IPv6/256 bits MEMBW: Number of Nodes per Level 107
Table B.3: VLC IPv6/1024 bits MEMBW: Number of Nodes per Level 108
Table B.4: SLC IPv4/256 bits MEMBW: Number of Nodes per Level 108
Table B.5: V/SLC IPv4/256 bits MEMBW: Number of Nodes per Level 108
Table B.6: SLC IPv6/256 bits MEMBW: Number of Nodes per Level 109
Table B.7: Simplified Range Trie VLC IPv4/256 bits MEMBW: Number of Nodes per Level 109
Table B.8: Simplified Range Trie VLC IPv6/256 bits MEMBW: Number of Nodes per Level 110

Table D.1: List of Communication Commands 113

List of Acronyms

ASIC	Application Specific Integrated Circuit
BU	Bottom-Up
CIDR	Classless Inter-Domain Routing
CMPWIDTH	Comparator Width
CP	Common Prefix
CPU	Central Processing Unit
CS	Common Suffix
FPGA	Field Programmable Gate Array
GE	Greater or Equal
GPU	Graphics Processing Unit
HTX	Hyper Transport (bus)
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISA	Instruction Set Architecture
L	Less
MEMBW	Memory Bandwidth
MSB	Most Significant Bit
RT	Range Trie
SLC	Single-Length Comparisons
TD	Top-Down
V/SLC	Variable per node/Single per comparator -Length Comparisons
VHDL	VHSIC Hardware Description Language
VLC	Variable Length Comparisons

Acknowledgements

This thesis is the result of personal hard work and the help and support of many people. First, I would like to deeply thank my supervisor Ioannis Sourdis for the opportunity to work on the range trie project and his guidance through all the phases of its development. Also, my gratitude goes to Sri Harsha Katamaneni for his help with understanding the initial codes, and a big thanks to Dion van Adrichem and Dimitrios Mastrogiannopoulos for the nice collaboration and the hours we spent together in the lab.
I would also like to express my sincere gratitude to my parents Spyros and Maria, my sister Eleni, my girlfriend Zeng, and all the people close to me for their good care and support in every aspect during my studies and my thesis in particular.
Last but not least, I owe special thanks to Lilian Voudouri Foundation which deemed me worthy of scholarship and supported me throughout my studies in TUDelft.

Charalampos Mastrogeorgopoulos
Delft, The Netherlands
14th of October, 2011

1 Introduction

Everyday people make use and also produce information in the context of their professional, social or cultural activities. The quality and quantity of this information constitutes a core element which distinguishes the modern societies from those of the previous centuries. However, information not available to those who need it defies its purpose, thus the need for efficient means of communication is essential.

Nowadays, Internet is widely considered the largest source of information. Though in reality a computer network, its size is enormous and continues to increase rapidly, both in number of users [1] and network traffic [2], but also in the relevant infrastructure [3]. Figures 1.1, 1.2, 1.3 present the relevant statistics of Internet users globally from 1995 to 2010, the total amount of Internet traffic in petabytes per month, and the number of Internet hosts respectively.

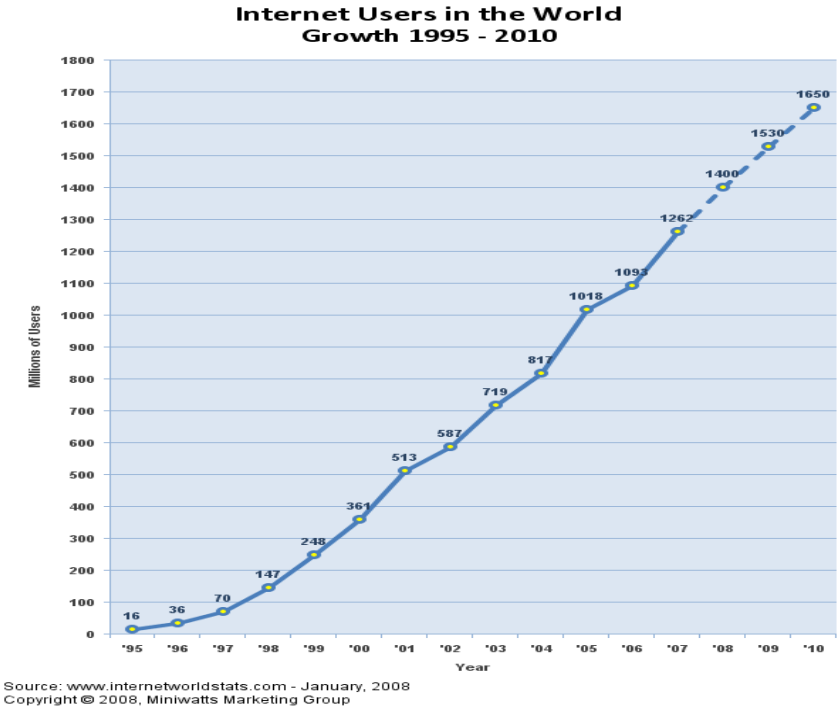

Figure 1.1: Internet Users Growth

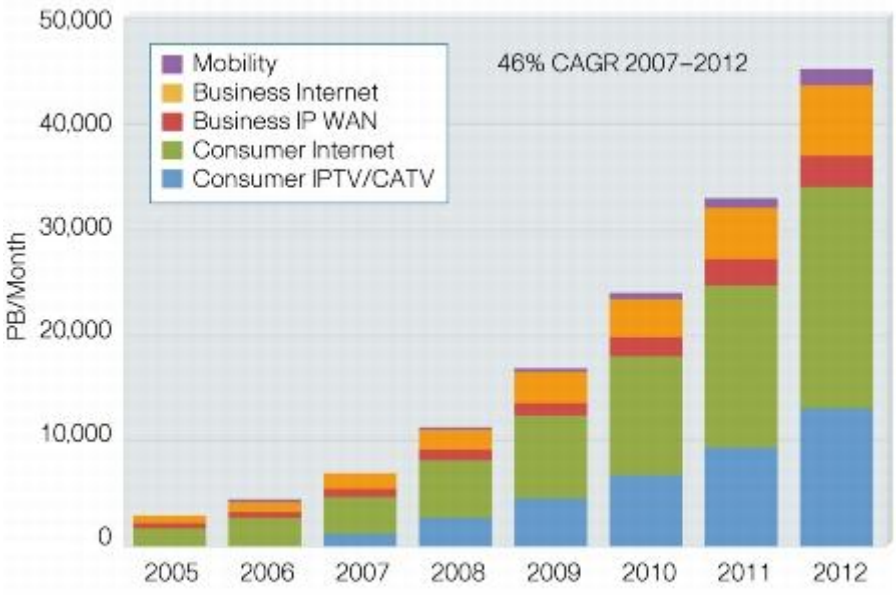

Figure 1.2: Internet Traffic Growth

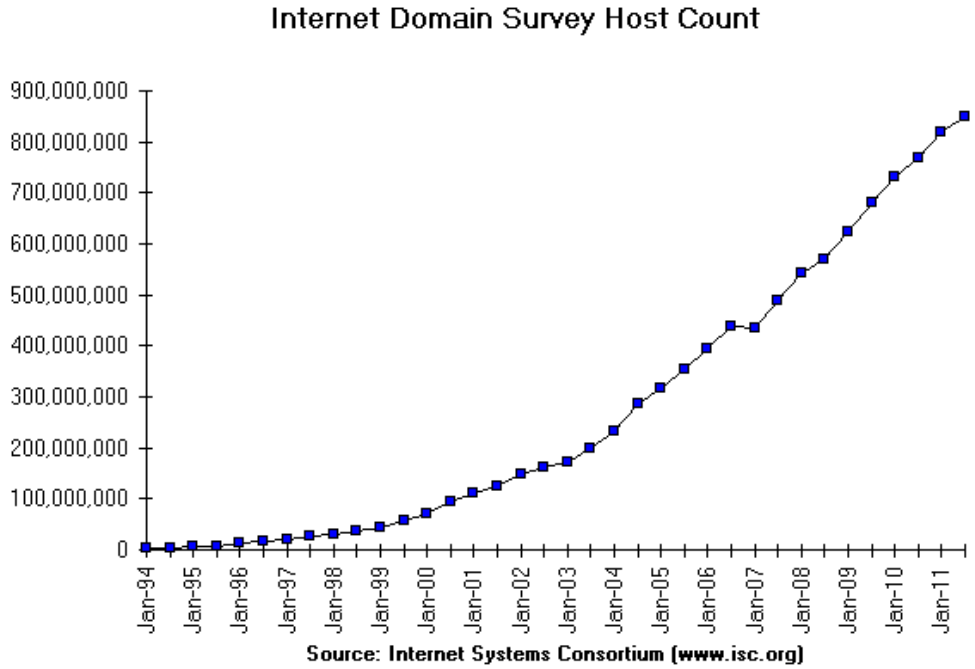

Figure 1.3: Internet Host Count Growth

Information is organized and transferred through the Internet in *packets*, the number of which depends on the amount of data needed to be transferred. Each packet consists of the *header* section which contains the *source* and *destination address* along with other metadata, and the *payload* section which contains the real data.

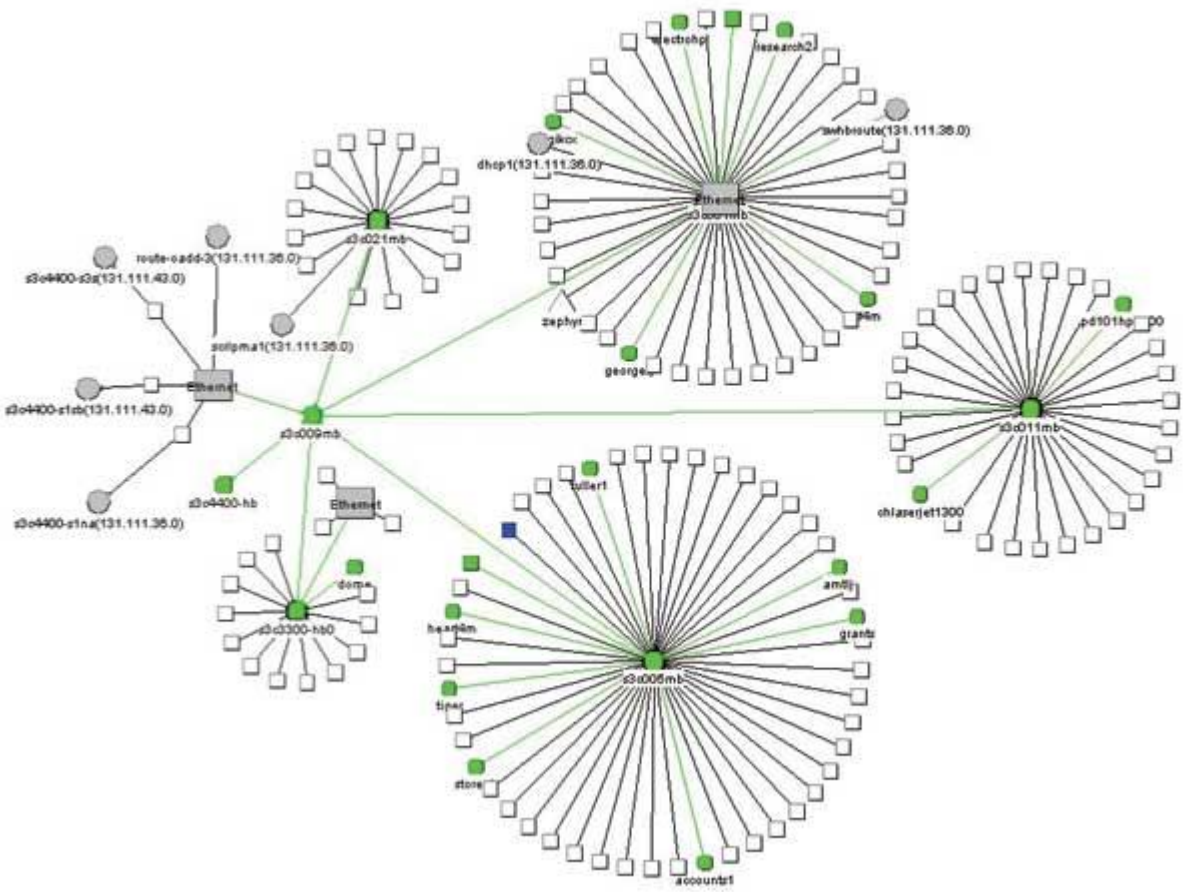

Figure 1.4: Example Network Topology

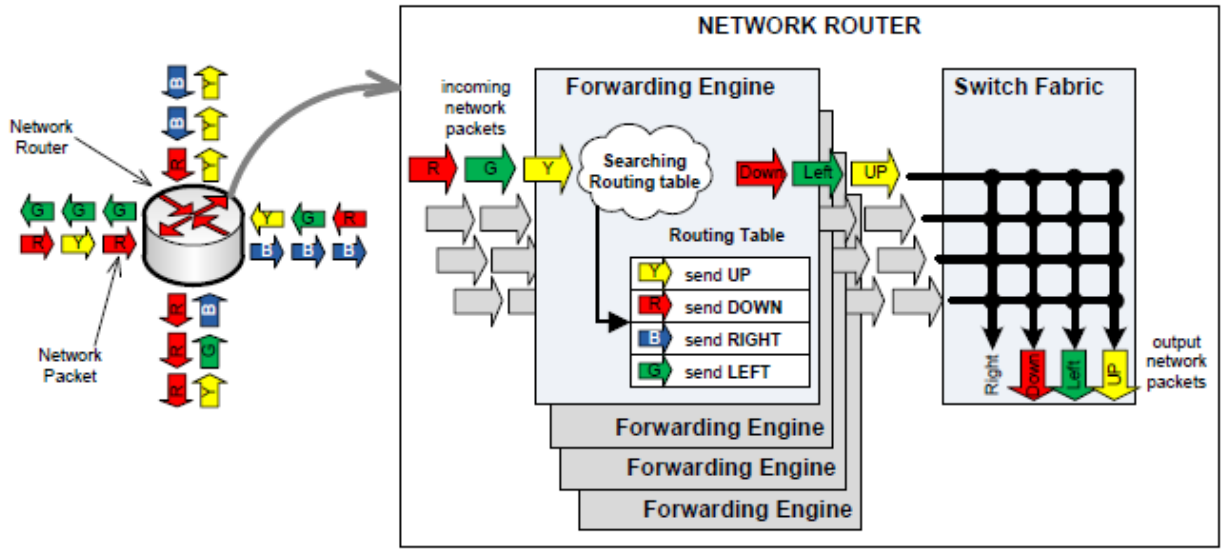

Figure 1.5: Network Router

The packets are routed in the network through a number of links and intermediate nodes (see figure 1.4). In every node a packet arrives, an output port is selected, which is connected to a link and subsequently to the next node, and this way the packet is moving towards its destination. This forwarding process is performed by the router and consists of two parts: the *forwarding engine* and the *switching element*. The purpose of the forwarding engine is to determine the correct output port based on the destination address and a *routing table* stored in the node. The destination address functions as input to this routing table which contains output ports/actions to be taken for all input addresses. The process of searching in the routing table is called *address lookup.* The switching element performs the actual packet forwarding through the port determined by the forwarding engine. Figure 1.5 (source [4]) presents the schematic concept of a network router.

The correct output port/actions are stored in the routing table in the form of *prefixes*. Example of an IPv4 prefix represented in binary is the following: 00110100xxxxxxxxxxxxxxxxxxxxxxxx, where 'x' denote "don't care" bits. We observe that the prefix has an initial part of fixed bits, followed by a part of "don't care" bits. The number of fixed bits is called *prefix length.* This representation implies an address range bounded by the lowest and highest possible value of the given prefix. For the example of the previous prefix these bounds create the range [00110100000000000000000000000000, 00110100111111111111111111111111], which is an alternative representation of the prefix as *address range*. For each prefix an output port/action is stored. These prefixes cover the whole address space.

1.1. Address Lookup

When a lookup is performed for a destination address, we try to determine the matching prefix. In the case of prefix/prefix length representation a prefix matches to the destination address if the fixed bits are the same. In the case of address range representation the prefix matches if the destination address belongs to the address range. However, possibly more than one prefix match with a given address. In this case matching prefix is considered the one with the greater prefix length, or equivalently the narrower address range. For this reason, the address lookup is equivalently also called *Longest Prefix Match.*

An example of an IPv4 address in practice can be seen in figure 1.6. A prefix can be noted as an address followed by a '/' and the prefix length. For the case of figure 1.6, a prefix based on this address could be the 172.16.0.0/16. Notice that the .0.0 suffix is of no interest, it could have any value without affecting the prefix since the prefix length is 16, and thus the remaining 16 bits cannot modify the prefix. Moreover, figures 1.7 and 1.8 show a schematic representation of an IPv4 routing table with three output ports and a lookup/longest prefix match performed for the given destination address 192.168.1.23 respectively (source [5]).

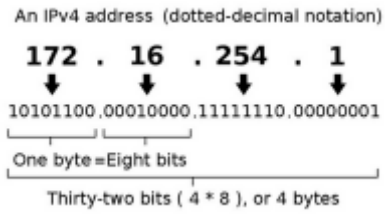

Figure 1.6: IPv4 Address

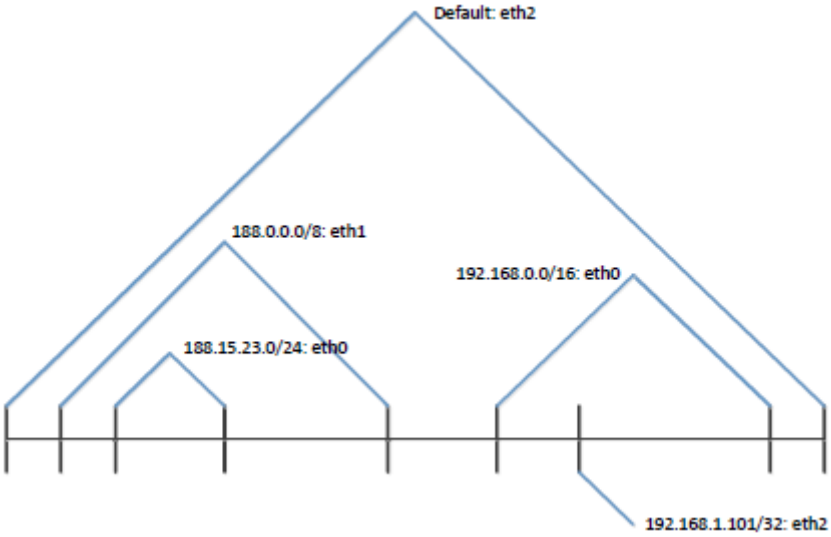

Figure 1.7: IPv4 Routing Table

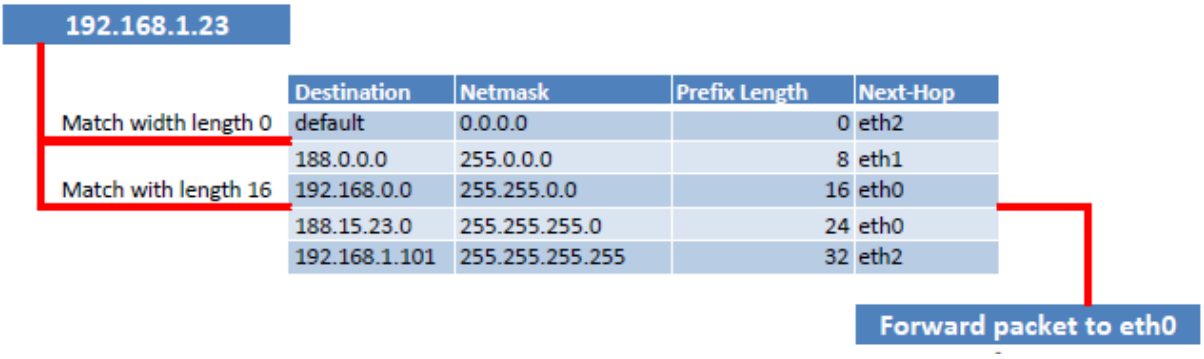

Figure 1.8: IPv4 Address Lookup

1.2. The Problem

As we mentioned earlier, Internet traffic has increased drastically the last years [2]. This means that each network router has a larger amount of packets to route through time. In order to handle this increased traffic, routers have to increase their throughput. Moreover, the number of nodes has also increased [3], and depending on the network structure, this may mean more interconnections per node, which subsequently increases the size of the routing tables. Certain algorithms (see range trees among others in chapter 2) do not scale well with it and thus the latency per lookup in the nodes increases significantly. Taking into consideration that the average packet needs to be forwarded through a higher number of nodes than before, the end-to-end delay increases even further. This is aggravated by the transition from IPv4 to IPv6. Expanding the addresses from 32 bits to 128 bits signifies an exponential increase of the address space and also lengthening of the prefixes. This results in a proportional latency per lookup increase for algorithms like the various forms of tries (binary trie, multi-bit trie and others, see chapter 2 for more details).

The forwarding engine constitutes the bottleneck of all the current router designs. According to our current knowledge, *none of the methods of address lookup scales well from latency perspective*. Either with the increase in the number of prefixes or the increase in the address width, resulting in progressively increased latency per node, and adding the greater number of nodes until a packet reaches its destination, Internet faces the threat to become slower due to its sheer size. Moreover, the increase in the number of prefixes stored in the routing tables results in proportional increase of their size. Beyond a possible complexity increase of the address lookup algorithms, a certain fact is that they will require more resources, mainly memory which will consequently lead to greater size, but also higher power consumption and need for stronger cooling. All these have a significant financial impact on designing, the maintenance and operation of the routers.

Moreover, depending on the algorithm, performance of address lookup procedure may be deteriorated when the relevant structures are updated (see imbalanced range trees for example), further contributing to the problem. For all the above mentioned reasons much research is performed on the field of innovative address lookup algorithms. In addition, certain parts of current and new algorithms is attempted to be sped up by implementing them in hardware directly, either on an FPGA or an ASIC. The main advantages of a hardware implementation in comparison to a software implementation is that parallelization can be exploited to the highest degree, and also complex functions can be implemented directly in bit level without the restrictions accompanying a general purpose CPU with fixed ISA. In both cases, these advantages may potentially decrease significantly the number of cycles needed per lookup.

The range trie constitutes an innovative algorithm for address lookup which can potentially solve the problem of the increased latencies. It shares some similarities with a range tree, but its strong point and the basic idea behind it is a series of optimizations which promise to offer greater scalability than any of the algorithms presented in chapter 2. This scalability refers not only to the decreased latency achieved by the small number of levels for a large number of prefixes, but also to resources utilization, with main concern the required memory, but also the processing elements too.

1.3. Thesis Objectives/Work

The work presented in this thesis is separated into two distinct parts. The first part consists of the configuration file generation software. The range trie scheme was firstly introduced in [6] and subsequently a first implementation has been demonstrated in [4] and in [7] where the focus was on the hardware and the software respectively. The latter was accompanied with an initial implementation of the configuration file generation software. This version has been slightly modified in [8], serving as input to the prototype VHDL implementation of the range trie. Due to certain issues presented in detail in [5], that design was partially functional in simulation level only, but not when ported on a real FPGA.

The first part of the work involved in this thesis consists of the debugging, verification, refinement, and improvement of the configuration file creation software. Specifically:

- A number of bugs/edge cases have been detected and corrected resulting in reliable and accurate software.
- The correctness of the resulting configuration file has been verified through the practical testing performed in [5] on the HTX platform range trie implementation, either through simple inspection of the results or through the software created for the second part of this thesis.
- The software has been refined, being generic for various parameters, enabling various modes of operations and permitting easy adaptation to various hardware implementations.
- The improvement refers to a more efficient way of processing than the initial. As it will be explained in chapter 3, the software has been modified in order to process the levels in a dynamic way permitting a more efficient memory management system. This, along with the use of hard disk space for key intermediate results, has solved the problem of the insufficient resources for large routing tables.

The second part consists of the support software for the HTX implementation. This enables an automated initialization process of the HTX platform with the range trie, and also constitutes a form of formal verification of the configuration software itself based on a variety of extracted information. These data has been also used in the process of maintaining the range trie structure (see chapter 4).

1.4. Thesis Overview

The rest of the thesis report is organized in the following way. Chapter 2 contains a variety of address lookup algorithms along with an extensive overview of the range trie structure, its optimizations and properties/characteristics. Eventually, the various methods of construction of the range trie will be presented.

In chapter 3 we will present the work related to the configuration file generation software and the additions/modifications performed on it. As closure of the chapter we will present the results for various routing tables/hardware characteristics and we will extract useful information about the characteristics of the software and the resulting configuration of the range trie.

Chapter 4 describes the support software for the HTX platform range trie implementation. A detailed presentation of its principle of operation will be given along with information about the method it follows in order to collect, process and utilize all the relevant data. Result of this procedure is the verification of the correctness of the configuration file creation software and the respective hardware implementation, serving at the same time as support software for initializing and maintaining the range trie in the hardware.

Chapter 5 constitutes the summary of this thesis in which we will conclude with an analysis of the results presented in the previous chapters along with proposals for future work and possible improvements.

2 Background

As we mentioned in the previous chapter (see section 1.2), the bottleneck of the processing performed in the router is located in the forwarding engine and specifically in the address lookup operation. Desirable characteristics for an efficient address lookup mechanism are high performance in terms of throughput and latency, as well as low resources mainly in terms of the required memory, but also processing resources and power consumption. Moreover, this mechanism must be flexible enough to permit easy updating/maintenance. Lastly, considering the increase in the amount of traffic data, the faster Internet connections, the enlargement of the lookup tables size, and the transition from IPv4 to IPv6, it is obvious that in order for address lookup schemes to be able to keep up with the higher requirements, they have not only to be adequate based on the current parameter sizes, but they also have to exhibit good scalability. This will allow them to remain efficient with even higher requirements and larger amount of data to handle with relatively small increase in the resources utilization.

The range trie scheme, initially proposed by Ioannis Sourdis et al. in [6], promises to meet all the above mentioned requirements, and the work performed for this thesis is fully related to it. However, it is highly preferable to present it in the context of other currently available schemes. This way the advantages of the range trie scheme will become apparent. These advantages constitute the base in order to explain the experimental results presented in section 3.2. Before analyzing core address lookup algorithms closely related to the range trie we will briefly present some basic addressing schemes.

The initial address lookup schemes applied in order to solve this problem were based on using different but limited addressing modes. The first used in practice (in Internet Protocol (IP) RFC 971) was called *classful addressing scheme* [9] because it followed a simple address allocation scheme and divided the addresses in three classes, A, B, and C , as it can be seen in figure 2.1. As it is apparent from this figure, the address is split in two parts, network part and host part, and each class has the separator of these two parts in different position. Performing lookup was simple: first the class was identified and subsequently exact prefix match was performed on the respective prefix table using binary search or hashing algorithms. With the rapid Internet growth however, this scheme, based on 32-bits long addresses, led to address space exhaustion. The *classless inter-domain routing* (CIDR) firstly presented in [10] eventually replaced it. This addressing scheme is essentially the current addressing scheme, which is based on a variable length prefix annotated with the prefix length. Figure 2.2 demonstrates an example of an IPv4 version of it (32-bits long addresses). The address space is used much more efficiently with this scheme, and it also allows IPv6 addresses (128 bits long). However, it requires longest prefix match which complicates significantly the address lookup operation. All the address lookup schemes presented in the rest of this chapter refer to CIDR addressing scheme.

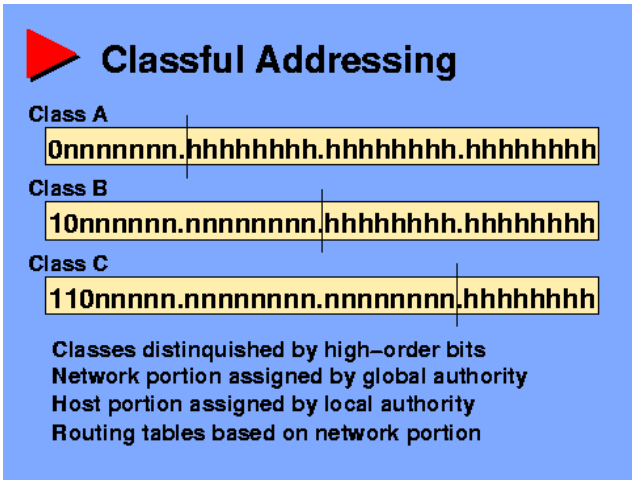

Figure 2.1: Classful Addressing

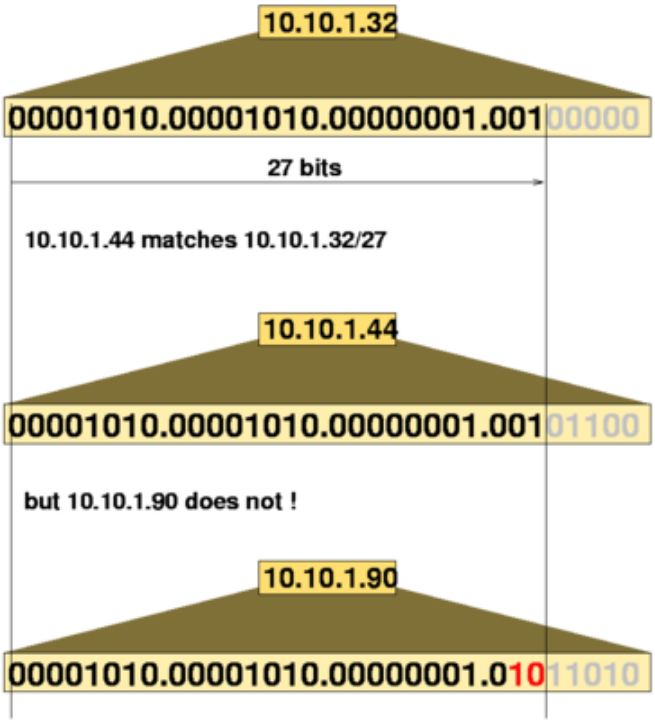

Figure 2.2: CIDR Example

In the rest of this chapter we will briefly present other related address lookup algorithms in section 2.1, and subsequently the mechanics of the range trie structure in 2.2. We will conclude by summarizing the techniques in section 2.3.

2.1. Address Lookup Related Algorithms

In this section we are going to present the state of the art concerning algorithms related to the address lookup operation.

These algorithms can be classified in two general approaches, depending on how they approach the problem. Some approaches consider it and try to solve it as a longest prefix match problem, while others consider it an address range lookup problem. We have to note that these two problems are equivalent, the difference lies in the address table representation.

The *prefix address* representation is denoted by a string of binary digits followed by an asterisk ('*'). The asterisk represents a string of "don't care" digits until the total string length is the appropriate one for the particular type of address (32 bits for IPv4, 128 bits for IPv6). This representation is the direct translation of the CIDR addressing scheme. The typical way of storing this representation in digital memory is by a doublet of prefix/prefix length, as it has been presented in figure 2.2 (see prefix denoted as 10.10.1.32/27). However, it is not unique, with an alternative being prefix/prefix mask (the equivalent of the prefix/prefix length 10.10.1.32/27 is 10.10.1.32/255.255.255.223 in prefix/prefix mask representation).

The alternative representation is based on *address ranges*. The address ranges are defined by their bounds, with the typical representation having the left bound included in the range and while the right is not included. Of course, these representations are also equivalent, and it is relatively easy to transform the one representation into the other.

Table 2.1 shows an example of a lookup table containing both these equivalent prefixes representations. This lookup table will be also used as basis in the example of the algorithms presented in the following sections. It is useful to note at this point that any arbitrary prefix/prefix length representation denotes a valid prefix, while two arbitrary bounds (left included, right not included) denote a valid prefix only when their difference is an integer power of 2. Moreover, the last N bits for the low bound (and consequently for the high too) must be zeroes, where N is that power ($HighBound - LowBound = 2^N$).

ID	Address Range	Address Prefix
I1	[000000,100000)	0*
I2	[010000,010010)	01000*
I3	[011000,100000)	011*
I4	[100000,1000000)	1*
I5	[100000,101000)	100*
I6	[110000,110100)	1100*
I7	[110100,111000)	1101*
I8	[111000,111100)	1110*
I9	[111100,1000000)	1111*

Table 2.1: Example Lookup Table

However, the representation facilitates different problem approaches. According to the taxonomy presented by Ruiz-Sanchez et. al. in [11], the current address lookup schemes perform either *"search on length"* or *"search of value"* to solve the problem. Figure 2.3 presents this notion schematically (source [4]). Prefix address representation of the lookup table facilitates a "search of length" approach, while an address ranges representation facilitates a "search of values" approach. Solutions belonging to any of these categories require increased memory when the size of the lookup table increases. From performance perspective, "search on length" approaches generally do not scale well with an increase of the address width (depending on the method, the latency can increase even up to proportionally to the increase in the address width) while they are little or not at all affected by an increase in the size of the lookup table. "Search on values" approaches generally are less affected by an increase in the address width (only the comparisons are slower, which is usually small portion of the whole processing time needed) but scale worse with an increase in the size of the lookup table.

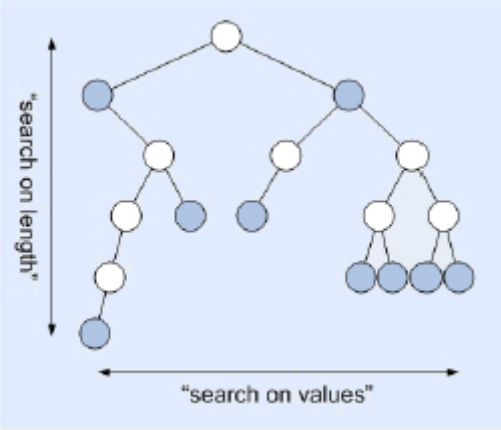

Figure 2.3: Address Lookup Approaches

In the rest of this section we will briefly present the most common approaches from these two categories. Specifically, sections 2.1.1 through 2.1.6 describe typical search on length algorithms, while sections 2.1.7 and 2.1.8 describe search on value algorithms. Lastly, this section will be concluded by a brief description of a probabilistic algorithm in 2.1.9 and of hardware oriented algorithms in 2.1.10.

2.1.1. Binary Trie

The *trie* structure (trie from re*trie*val) is a data structure specifically created in order to perform search on the elements stored in it, and in our case to perform longest prefix match. It highly resembles a classic tree structure, but it is distinguished from it by the fact that results of the search operation can be obtained from intermediate nodes too, not only the leaf nodes.

The simplest version of a trie in order to perform longest prefix match is the *binary trie*. Figure 2.4 (source [4]) shows the binary trie corresponding to the lookup table represented by the prefixes in table 2.1. Each node of the trie has a maximum of two children. The trie is traversed based on the

destination in a bit-by-bit basis, with each of its bits to determine the binary branching (where the name comes from) to the node of the next level [11]. No bit counter is needed since for each branching only the next bit is always examined. When the search is at the Nth node of the trie, this means that *N-1* bits have been already used in order to traverse the trie up to that point and the Nth bit will determine the next branching. Some of the binary trie nodes correspond to prefixes, while some others not, with leaf nodes always containing prefixes. The search concludes when all the bits of the destination address have been used to traverse the trie or when during branching a non-existent node is selected. In both cases, the longest matching prefix is the last one contained in the nodes traversed during the search in the trie. In figure 2.4 the traversal of the binary trie with 01000* as destination address is indicated by the bold line.

Binary Trie represents the simplest form of "search on length" approach, and it is characterized as sequential since the longest matching prefix is attempted to be determined starting from the shortest and searching for matching prefixes with length increased by one per each step. Simple worse case analysis indicates that a search needs *O(W)* steps of bit comparison, which means poor scaling with address width. Moreover, the binary trie is generally imbalanced, leading to high memory requirements.

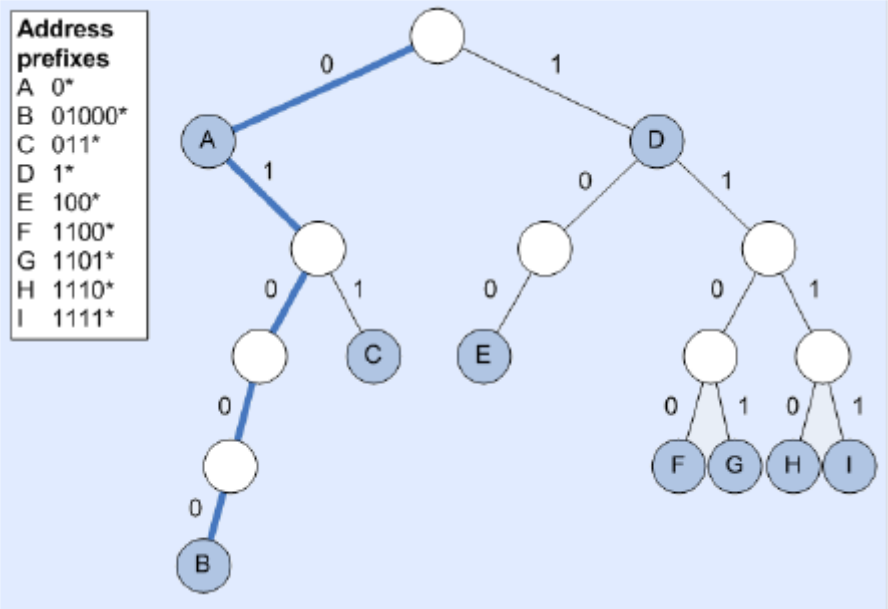

Figure 2.4: Binary Trie

2.1.2. Path-Compressed Trie

An improvement of the binary trie is the *path-compressed trie*, also a sequential "search on length" algorithm. It was firstly used in the PATRICIA scheme by Morrison [12], and an improved version was presented later by Sklower [13] . As it can be seen if figure 2.4, the path which leads to prefix B is long and dedicated to only one prefix. This leads to memory waste and multiple levels just in order

to inspect one single prefix. Path-compressed tries represent a modification of the binary tries by removing these long singular paths through a technique called *path-compression*.

This technique ensures that all the non-leaf nodes of the trie have always two branches by ignoring for each path the common bits and inspecting directly the bit that leads to branching and thus distinguishes the different paths. This reduces the length of these paths and leads to lower node count. However, the level of the node, in contrast to the case of the binary trie, does not indicate the bit to be inspected for the next branching, so this information has to be stored in the node. Moreover, by ignoring some bits we cannot ensure that the destination address matches to the prefix corresponding to the prefix node. For this reason the prefix has to be stored in the prefix nodes and a simple prefix match check is performed every time that node is visited. If the check is unsuccessful then the traversal stops and the longest matching prefix is the last successfully checked during the traversal of the trie. If the check is successful and the node is leaf node then that prefix is the longest matching prefix, otherwise the search continues by traversing the trie deeper and searching for longer matching prefixes.

Figure 2.5 (source [4]) shows the path-compressed trie corresponding to the lookup table represented by the prefixes in table 2.1. As we can observe, all the non-leaf nodes contain the bit to be inspected for the next branching, while the prefix nodes also contain the prefix itself. The trie traversal for the destination address 01000* is indicated by the bold line. In both prefix nodes A and B simple prefix check is successful, thus B as leaf node is the longest matching prefix. In contrast, The destination address 00000* would follow the same exactly path, but with prefix check at node A successful and at node B unsuccessful, thus A is the longest prefix node as the last successfully checked prefix.

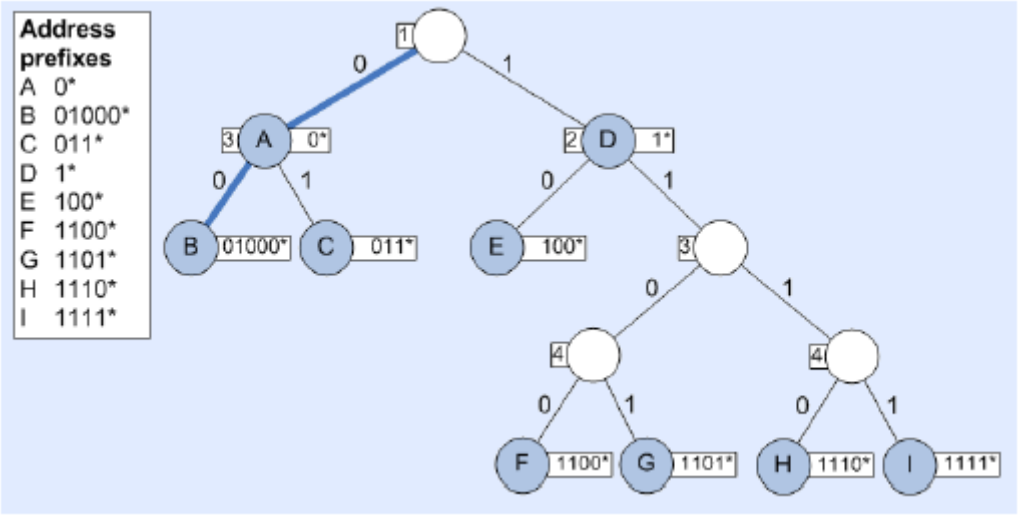

Figure 2.5: Path-Compressed Trie

However, even though the path-compressed trie is on average shorter than a binary trie, worst case analysis indicates that certain paths may not be able to be compressed, thus a search needs also *O(W)* time even for an efficient implementation. As a result, this approach also does not scale well with the address width. All of the following "search on length" algorithms have as main goal to improve this worst case time.

2.1.3. Multi-bit Trie

The *multi-bit trie* represents an alternative modification of the binary trie which allows a trade-off between, on the one hand the number of levels and consequently the search time (latency), and on the other hand the memory requirements. At each level, instead of examining for each node only one bit in order to determine the correct branching as it happens in the binary trie, multiple bits are examined at once. The number of these bits is called *stride*. If nodes at a level have a stride *N*, this means that these nodes can have a maximum 2^N number of children.

Thus, bigger stride signifies a decreased number of levels and higher branching factor. However, multi-bit tries do not support all the prefix lengths. For this reason the not supported prefixes have to be expanded to the next longer supported prefix length. Considering that k_1 is the length of a not supported prefix and k_2 is the length of the next longer prefix length supported, then this prefix needs to be expanded to $2^{(k2-k1)}$ prefixes of length k_2. This leads to higher memory requirements and constitutes the trade-off for the increased performance.

The strides do not need to be of the same size for each trie node. Multi-bit tries can have fixed strides or variable per node. Figure 2.6 (source [4] but modified in order to match correctly with the multi-bit trie structure) shows the multi-bit trie corresponding to the lookup table represented by the prefixes in table 2.1. As we can observe, the root node has stride two, and for this reason prefixes A and D with length one have each to be expanded into two prefixes of length two. Similarly, C has length three but the corresponding parent node corresponds to length two and has stride also two, thus C is not supported directly and it is also expanded to two prefixes of length four. Again, multi-bit trie traversal for the destination address 01000* is indicated by the bold line.

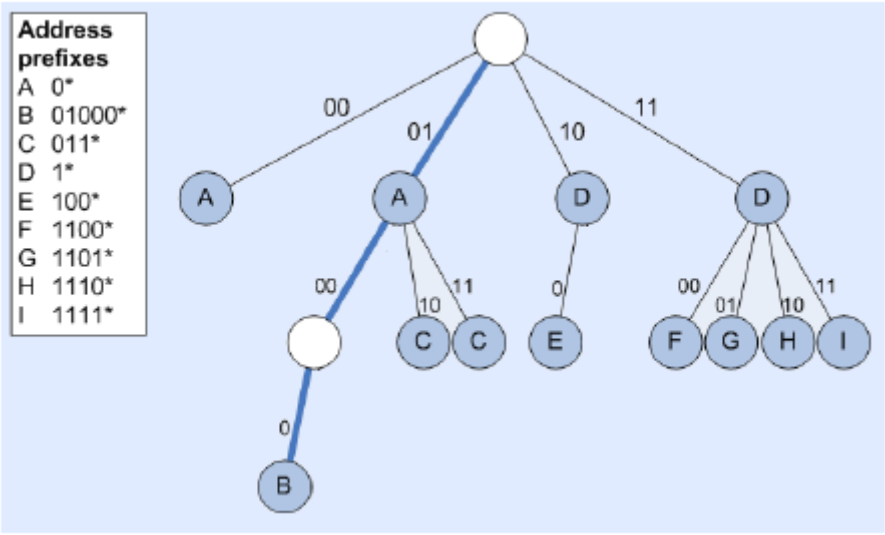

Figure 2.6: Multi-bit Trie

For a fixed-stride multi-bit trie with stride *k* a simple worst case analysis indicates that a search needs *O(W/k)* time, thus we observe that multi-bit tries offer a constant factor improvement to the performance. Multi-bit tries also represent a sequential "search on length" approach.

2.1.4. Level-Compressed Tries

The *Level-Compressed Tries* (*LC-tries)* attempt to combine the advantages of the path-compressed tries presented in section 2.1.2 with those of the multi-bit tries presented in section 2.1.3 in order to achieve even better average case address lookup time, worst case time equal to those of the of the multi-bit tries (*O(W/k)* where k the stride), and lower memory requirements taking into account the gain in performance.

The method to construct a level-compressed trie is by transforming the initial binary trie. Initially, *k*-levels full binary sub-tries are recursively substituted by a *k*-stride node, and subsequently path-compression is performed to the resulting trie. Even higher performance is gained if the criterion about full binary sub-tries is loosened, which means that some branches may not exist and subsequently not full utilization of the *k*-stride node is permitted. However, this leads to memory waste and thus higher memory requirements. The fraction of the valid branches for a node compared to the sum of the potential branches for that node is called *fill factor 'f'* where *f* in (0,1]. The minimum allowed value of the fill factor *f* represents a parameter used during the construction of the LC-trie, with 1 signifying that only full sub-tries can be transformed to multi-bit nodes.

Figure 2.7 (source [4]) shows the level-compressed trie corresponding to the lookup table represented by the prefixes in table 2.1, where we can observe both sub-trie transformation to *k*-stride node (parent node of the leaf prefix nodes F,G, H and I) and path-compression in node A.

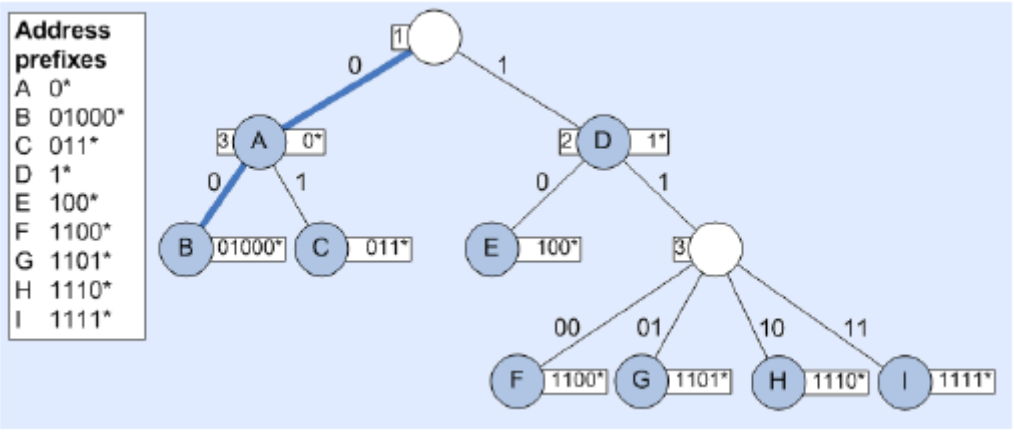

Figure 2.7: Level-Compressed Trie

Full expansion/compression by Crescenzi et. al. [14] and *Lulea algorithm* by Degermark et. al. [15] among others utilize this scheme to achieve fast search times.

2.1.5. Tree Bitmap

Tree bitmap, first introduced by W. N. Eatherton and Z. D. Dittia in [25], constitutes a further optimization of the multi-bit tries with emphasis on reducing the memory requirements thanks to more efficient storage of prefixes and nodes in the memory.

First step in the creation of the tree bitmap structure is partitioning the initial binary trie into *k*-stride multi-bit nodes. Subsequently, the existing children nodes of each node are stored sequentially in the memory, and the same applies to the information corresponding to the prefixes contained in the node. The information encoded in the node is:

I. A single pointer to the first of the children multi-bit nodes
II. A bitmap for the external pointers of the multi-bit node (external refers to the children nodes)
III. A bitmap for the internal pointers of the multi-bit node (internal refers to the prefixes contained in the node)

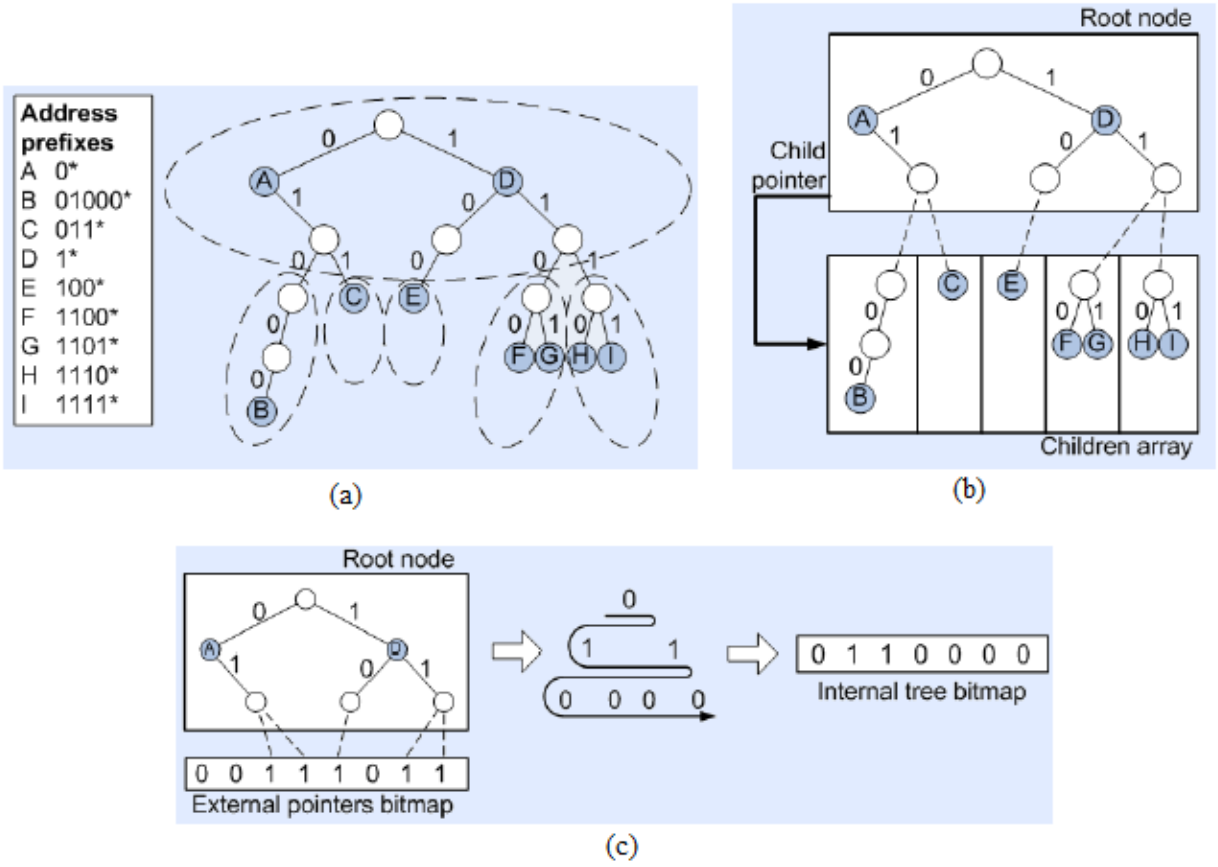

Figure 2.8: Tree Bitmap

Considering the fact that the children nodes are stored in the memory sequentially, the single pointer combined with the two bitmaps is sufficient for producing the next-level-node pointer and accessing the prefix data for the node itself. When the search terminates, a result pointer is used by the lookup

process in order to access the results array, and thus the action to be taken for the specific address is determined.

An example of the tree bitmap data structure can be seen in figure 2.8. The binary trie is separated into 3-stride multi-bit nodes as it can be seen in (a), and subsequently the five existing of the eight possible children nodes are stored sequentially in the memory, while at the same time a pointer is stored in the parent node and points to the first of the children nodes (b). In (c) the process of creating the external and internal pointers bitmap is demonstrated. Considering *k*-stride nodes, the external pointer bitmap, of size 2^N, contains 1 when the corresponding path leads to an existing child node or 0 otherwise. Similarly, the internal pointer bitmap, of size 2^N-1, contains 1 when the corresponding initial binary-trie node exists and is a prefix node, or 0 otherwise.

Tree bitmaps are flexible in the sense that allow different sizes of multi-bit strides and thus can be tuned to the specific memory characteristics. Possible optimizations of those are the *initial array optimization, end node optimization, split and segmented tree bitmaps, substitution of certain nodes with CAM nodes* (see section 2.10 for the latter).

2.1.6. Binary Search on Length

An alternative sequential "search on length" approach would be to arrange the prefixes in hash tables according to their length and perform exact prefix match starting from the longer prefixes and moving to the shorter sequentially until a match is detected [17]. Worst case analysis reveals that this does not reduce the worst case search time.

Waldvogel et. al. also in [17] propose instead a binary related to prefix length search. The first hash table to be searched is the one which holds prefixes of length equal to half of the address width. This splits the prefix space in prefixes of smaller length and bigger length. Depending on whether there has been an exact prefix match or not, the bigger length prefixes or the smaller length prefixes consists the new search space respectively, and the new hash table to be searched is the one which also splits the prefix space in the middle. This leads to only $O(\log_2 W)$ hash tables check instead of $O(W)$ as it is in the sequential case.

However, longer prefixes do not appear to the shorter hash tables, and the algorithm needs an indication whether it should move to longer or shorter prefixes. Except for longer prefixes which are covered by shorter ones, the rest would lead to erroneous behavior. For this reason "fake" prefixes (markers) of the real longer prefixes are inserted to the hash tables which will be inspected before the hash table containing the real prefix.

An example of this algorithm can be seen in figure 2.9 for the prefixes A – P visible on the top left of the figure. The binary trie is not used in reality, it is only depicted to demonstrate the fake prefixes inserted in the hash tables. These are indicated as M nodes, where the immediate shorter valid prefix and the fake shorter version of the longer prefix can also be seen. The fake stored in the marker is used to enable the algorithm to proceed with the search to longer prefixes, while the shorter valid prefix is used in case all the subsequent attempts to match with longer prefixes fail. When a lookup is performed for the destination address 1100000* the hash tables checked are: initially hash table for prefix length 4, then for 6 and last for 7. An exact prefix match is performed successfully in all of these tables, so the matching prefix is correctly the K. At hash table for prefix length 4 prefix F

already covers for the prefix K, but for length 6 there is no covering prefix, thus a fake prefix is inserted to guide the search. If a lookup was performed for the destination address 1100001*, the search at hash table for length 7 would have failed and the longest matching prefix would come from the marker as the immediate shorter valid prefix, in this case F.

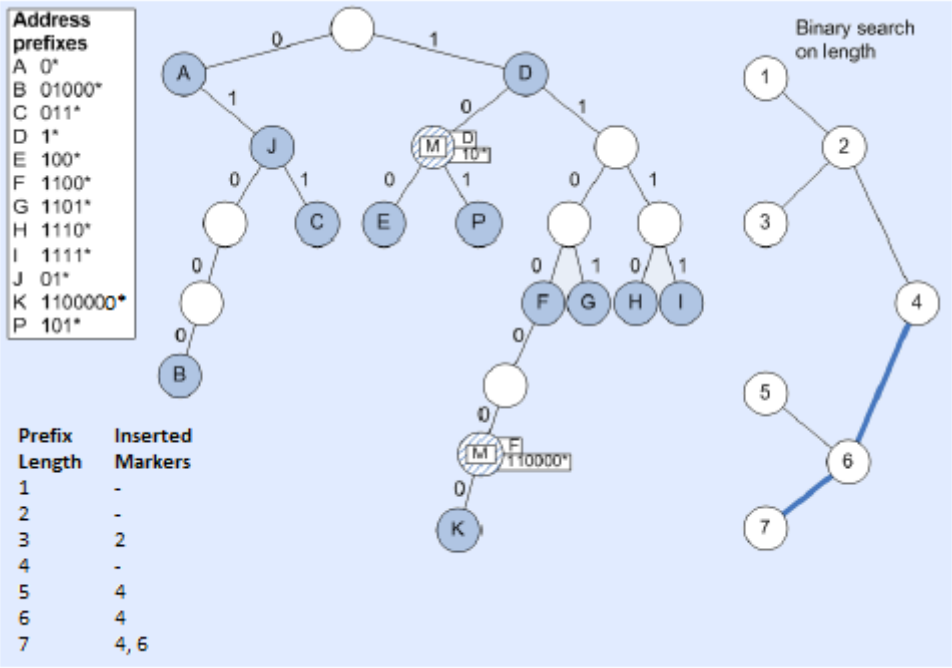

Figure 2.9: Binary Search on Length

The table on the bottom left of figure 2.9 shows at which hash tables it is necessary to insert fake shorter prefixes if covering prefixes do not exist already, based on the length of the prefixes.

2.1.7. Linear Search

The simplest address lookup approach is the sequential "search on values", or simply *linear search*. The destination address is attempted to be matched with all of the entries of the lookup table (usually represented as prefix/mask since the matching is then a straightforward comparison for the bits defined by the mask). From all the successful matches the longest prefix is selected as output. This scheme requires a number of comparisons independent of the address width, but proportional to the number of prefixes in the lookup table, thus it is unusable in practice. For this reason emphasis is given to non-sequential "search on values" algorithms.

2.1.8. Range Tree

The most common binary "search on values" approach is the *range tree*. Range trees are based on the address ranges representation of the lookup table. Instead of using the prefix itself, we expand it to its low (inclusive) and high (exclusive) bounds which define these ranges. The length of these bounds is equal to the length of the addresses, and based on this the range tree is constructed. At each of its nodes comparisons are performed between these bounds and the incoming destination address. The branching to the next level node is decided by the result of these comparisons. For low inclusive and high exclusive these comparisons are of the type less (L) or greater-or-equal (GE). The result is obtained by the leaves of the range tree which correspond to address ranges where the unique longest matching prefix is defined during the construction of the tree.

The number of comparisons per node further specifies the range tree. Range trees with only one bound per node split the address space into two intervals, and for this reason they are called binary range trees. Range trees with *k-1* bounds per node split the address space into *k* intervals and are generally referred as multi-way range trees [18]. The construction of the tree is arbitrary (meaning that there are many different ways to construct a tree which represents a given lookup table), thus with optimal selection of the bounds stored in each node the resulting tree can be balanced, leading to small tree depth. However, they tend to have higher memory access cost since for each node full bounds need to be transferred for comparison. Multi-way range trees represent a trade-off between lower tree depth and higher memory access cost.

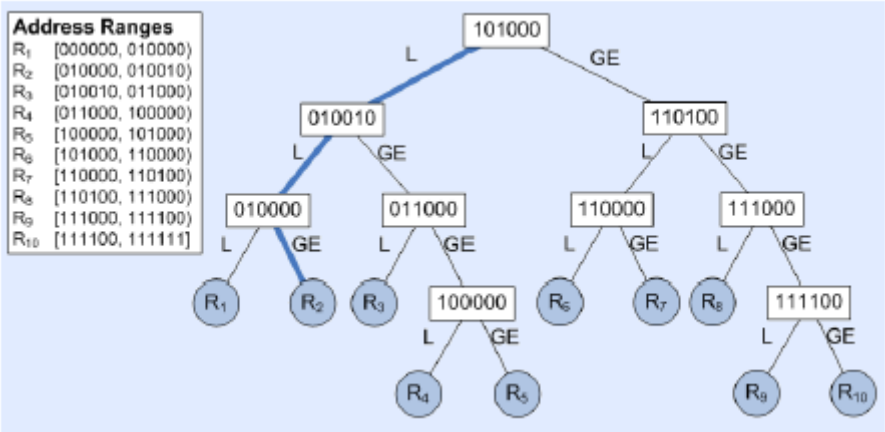

Figure 2.10: Binary Range Tree

Figure 2.10 shows a balanced binary range tree based on the lookup table shown at top left of the same figure. Note that since the bounds are low (left) inclusive and high (right) exclusive, the branching are decided by whether the destination address is less than the bounds, in which case it branches left, or greater than or equal to the bound, in which case it branches right. The equivalent balanced 4-ways range tree is shown in figure 2.11. From the figures we note that leaf nodes appear to levels with maximum difference one. This is the best case for an arbitrary number of bounds, and thus range trees with this property are considered balanced. In figure 2.10 the bold line indicate the path in

the range tree during the lookup of the destination address 010000, while equivalently the bold line in figure 2.11 represents the lookup of the destination address 110000.

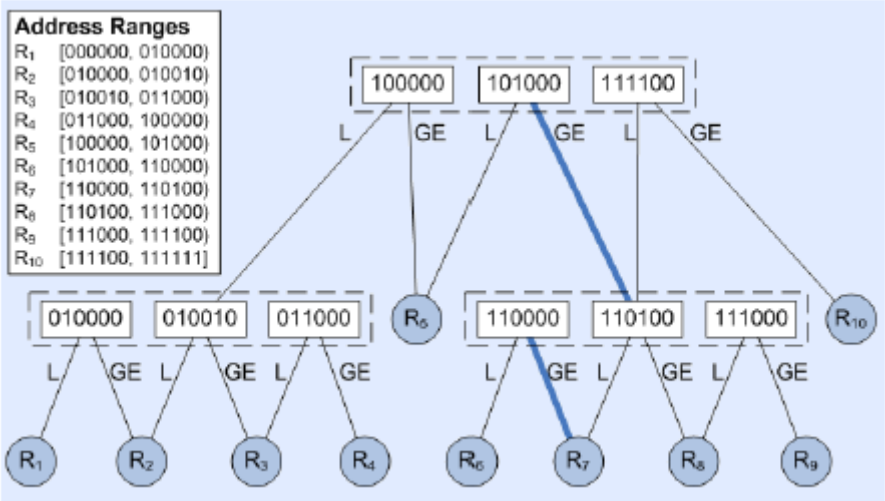

Figure 2.11: 4-ways Range Tree

2.1.9. Bloom Filters

Bloom filters were introduced by Bloom in [22] and constitute a probabilistic algorithm to test whether a certain element is part of a set or not. Using a *m*-bits long bit vector, initially all set to 0, and *k* hash functions able to map an element to one of the *m* bits of the bit vector, the Bloom filter for that set is constructed by feeding all the elements of the set into the *k* hash functions and setting 1 all the bits in the resulting positions. Subsequently, to check if a given element is part of the set, the element is fed into the *k* hash functions and thus *k* bit positions are obtained. If all these *k* bit positions are 1 then the element *probably* belongs to the set, while if even one of them is 0 then the element *definitely* does not belong to the set. As a result there is a certain probability for *false positive* stemming from the fact that the Bloom filter is set by the combination of all the members of the set. This probability is affected by the number of entries, size of the bit vector and number of hash functions.

Example of a Bloom filter can be seen in figure 2.12 (source [4]). Initially a, b, and c are fed into the 3 hash functions and the respective bits of the bit vector are set 1. Then we perform search whether elements c, d and f belong to the set by using the same hash functions and inspecting the resulting bit positions. For element d one of the three resulting positions contains 0, thus it does not belong in the set. On the other hand, c and f have all the resulting positions 1, but only c really belongs to the set, while f is false positive.

A more advanced approach is the *counting Bloom Filters*, introduced in [23], where the bit vector has been substituted by a counter vector of the same size *m*. At the cost of higher memory requirements this scheme allows the deletion of prefixes from the set by decreasing the value of the respective

counters, while the condition for matching transformed from “equal to 1” to “greater than or equal to 1”.

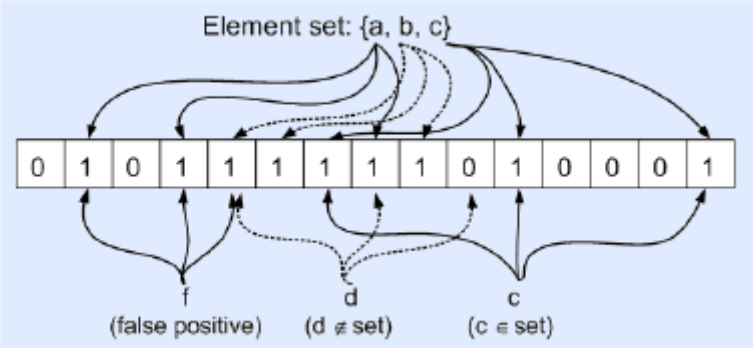

Figure 2.12

Dharmapurican et al. in [24] used this scheme to address the problem of address lookup. They created and programmed separate Bloom filters for each prefix length, and to perform a lookup the respective parts of the input address are queried in parallel. Subsequently, the matching hash tables are searched starting from the longest. Search terminates upon exact match (which detects the case of false positive) or if all hash tables have been searched.

2.1.9. Hardware Oriented Solutions

For sake of completion we will simply refer here some address lookup schemes which aim specifically at a hardware implementation, either in order to exploit the existing parallelism or to make use of special operations which can be optimized in hardware. Those available in literature are the *TCAMS* (see [19] and [20]), *IPstash* (see [21]) and *Pipelines* (see [26] and [27]). They generally achieve good performance at the cost of high resource utilization.

2.2. Range Trie

An ideal address lookup scheme would be required to exhibit the following characteristics as we mentioned in the introductory chapter (see section 1.2): high throughput to be able to handle the large traffic in the Internet nodes, low latency in order to decrease the end-to-end delay, and low memory/resources utilization for obvious economic and technological reasons. Moreover, it has to exhibit good scalability for all these characteristics with an increase in the size of the lookup tables and/or address width. Lastly, easy construction and maintenance of that scheme also represents a secondary desirable characteristic. The *range trie* structure first proposed by Ioannis Sourdis et. al. in [6] promises to fulfill this purpose in a higher degree than all the methods presented in the previous sections.

The range trie represents and intermediate approach between the range trees and the tries. For this reason it cannot be categorized in any of the "search on length" or "search on values" categories. Tries focus on "search on length" by performing *exact match in parts of the addresses*, and as a result they exhibit good scalability in reference to the size of the address table. Range tree structures focus on "search on values" by performing *comparisons on full addresses*, and thus exhibits good scalability in reference to the address width. The range trie structure attempts to assimilate both of their advantages in scalability by performing *comparisons on parts of the addresses.*

The structure and traversal of the range trie resembles that of the multi-way range tree (see section 2.1.8) by essentially performing comparisons on addresses and determining the correct branching from the result of these comparisons. However, a series of optimizations which are based on the concept of the tries allows comparisons of parts of the addresses to represent the implied full address comparisons. This signifies a better utilization of the available memory bandwidth since partial addresses need less bits to be represented and thus more of those addresses can be packed into a memory line. Each memory line corresponds to the data contained in a node, thus the increased number of addresses per memory line signifies higher branching factor for the nodes. The branching factor varies per node since the optimization can be more or less successful for the set of containing addresses, but overall it is higher than that of the range tree, leading to a smaller number of nodes and levels.

Figure 2.13 (source [4]) shows a typical range trie node schematically. A_1 to A_k are the k addresses stored in a given node (they are stored in a compressed form, the real addresses are only implied and not actually contained in the node memory line), while N_a and N_b are the bounds of the node itself (these are totally implied, information about them is not stored at all in the node memory line). These bounds and addresses partition the node address space into $k+1$ continuous intervals as it can be seen in the bottom part of the figure 2.13, all inclusive to the left bound and exclusive to the right bound.

It is important to note here that these intervals define the node bounds for the respective children nodes, the same way that the current node bounds are defined by the bounds of the parent node. However, there are some exceptions. Figure 2.14 shows schematically a 3-level range trie, with one root node, three nodes at second level and eight nodes at third level. As we can see from the two black arrows, the last interval defined at a node and the first interval defined at its next node may be represented by the same child node, in which case the bounds of the child node are the last address (bound) of the left parent node and the first address of the right parent node. Exceptions also are the nodes of the last level and the root node. The root node has no parent node but it covers the whole address space, and the last level nodes have no children nodes and thus these intervals represent address ranges which can be mapped directly to prefixes (meaning that if an address belongs to that address range there is no doubt about which is the longest matching prefix, in which case more comparisons would be needed).

Thus, from the above it is obvious that the union of the intervals corresponding to the children nodes covers the whole address range of the parent node. This, if applied recursively from the root node and combining it with the fact that the root node covers the whole address space we reach the conclusion that the union of the intervals defined in all of the nodes of a range trie level covers the whole address space. This can also be seen in figure 2.14 from the fact that the nodes at each level are consecutive (without gaps or overlapping) and cover all the possible addresses.

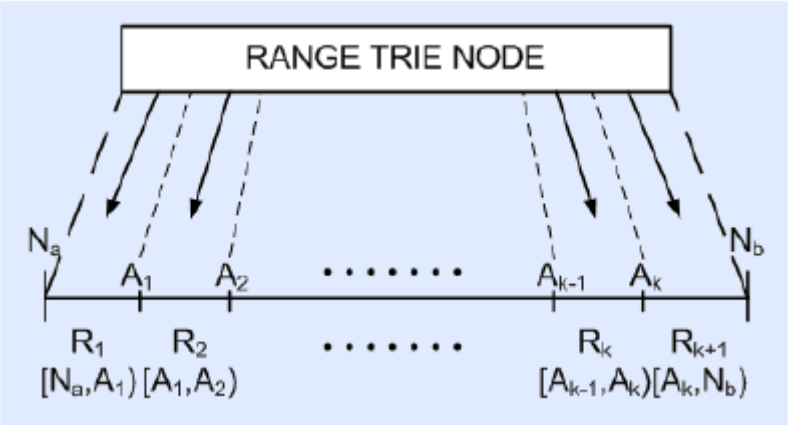

Figure 2.13: Range Trie Node

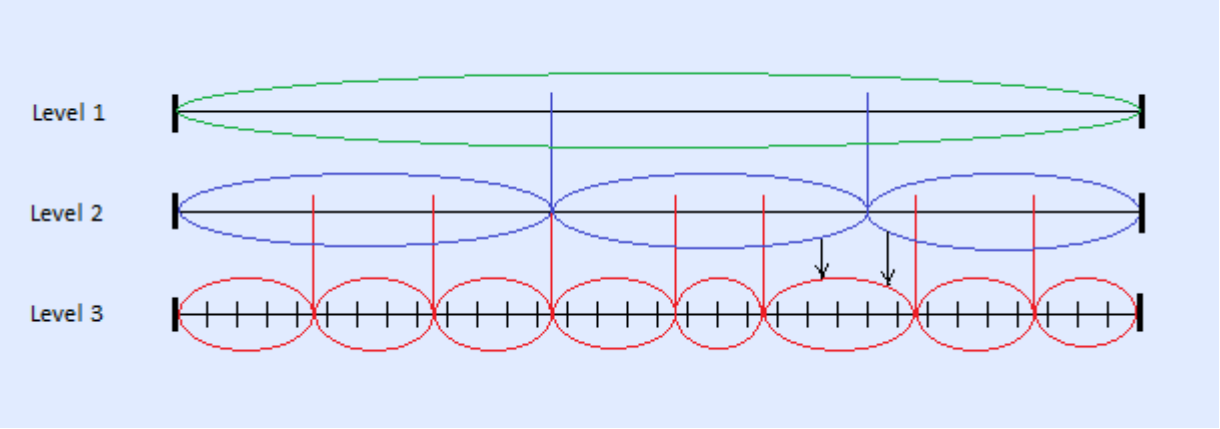

Figure 2.14: Bounds Passing/Child Node Sharing

The rest of this section is organized as follows: in section 2.2.1 and its subsections 2.2.1.1 to 2.2.1.5 the rules based on which the range trie is constricted will be described in concept. Subsequently, the range trie structure will be discussed in 2.2.2, while in 2.2.3 we will outline the configuration generation techniques for the range trie structure. Section 2.2.4 continues with the presentation of the components which support the longest prefix match operation, while section 2.2.5 concludes the range trie description by presenting the dynamic aspect of dynamic range trie structure.

2.2.1. Range Trie Concepts and Rules

The basis of the idea behind the range trie structure is the following set of observations:

- Nodes close to the bottom of a balanced range tree represent addresses densely located in the address space, which means that they are distinguished by their lower part (their least significant bits, or equivalently their suffix). These addresses are mostly defined by the prefixes.

- Nodes close to the root of a balanced range tree represent addresses sparsely located in the address space. These addresses are mostly defined by the algorithm (not so much affected by the prefixes) which means that if selected properly they can be distinguished by their higher part (their most significant bits, or equivalently their prefix).
- Nodes in the middle of a balanced range tree represent addresses with intermediate density in the address space (key property is that the density does not vary a lot for those nodes – small density variance) and thus they share characteristics from both of the above mentioned categories but in smaller degree. For this reason there is an area around the middle part of these addresses which can be used for their distinction.

The range trie structure tries to exploit these observations by performing comparisons of parts of the addresses and that utilize in a better way the given memory bandwidth by "packing" more addresses in the same memory line. As we mentioned in the previous section this increases the comparison values per node, and consequently the branching factor, leading to a smaller node count and shorter range trie structures for a given number of prefixes.

In order to exploit these characteristics and eventually minimize the comparison widths a set of five rules has been utilized for the construction of the range trie:

I. **Rule 1:** Omit the common prefix of the node
II. **Rule 2:** Share addresses' common prefix
III. **Rule 3:** Share addresses' common suffix
IV. **Rule 4:** Omit addresses zeroes suffix
V. **Rule 5:** Address alignment

Rules 1 and 4 discard completely part of the address, rules 2 and 3 store and process that part of the address only once for the whole node, while rule 5 actually adds a partial address which though permits all the other rules to be applied more efficiently. Rules 1-4 can be applied independently, but special care has to be taken for rule 5. If all these rules are applied properly the correctness of this theoretical structure is ensured (see [6]).

In the rest of this section these five rules will be presented in greater detail.

2.2.1.1. Rule 1 – Omit the common prefix of the node

The first and most straight forward rule concerning the prefix part states that, if all the addresses in the address range of the node share a common prefix part, then this prefix can be omitted and thus not be used for comparison. This limits the comparison width to the bits which are actually used to distinguish the different addresses belonging to the node address range.

Example of application of this rule to a node can be seen in figure 2.15. As we can see from the values of N_a and N_b, the node address range is the [FFFF0000, 100000000) or equivalently [FFFF0000, FFFFFFFF]. Note that N_a and N_b are not really stored somewhere in the hardware so we do not have representation problem for N_b as we would have if a bound had the value 100000000. From the above we can see that any address accessing the node is of the form FFFFxxxx, thus the first 16 bits are constant and can be omitted. As a result, as we can observe in figure 2.15, 16 bits are sufficient to

store the compare bounds. The branching process is exactly the same as in the case we compared the full addresses.

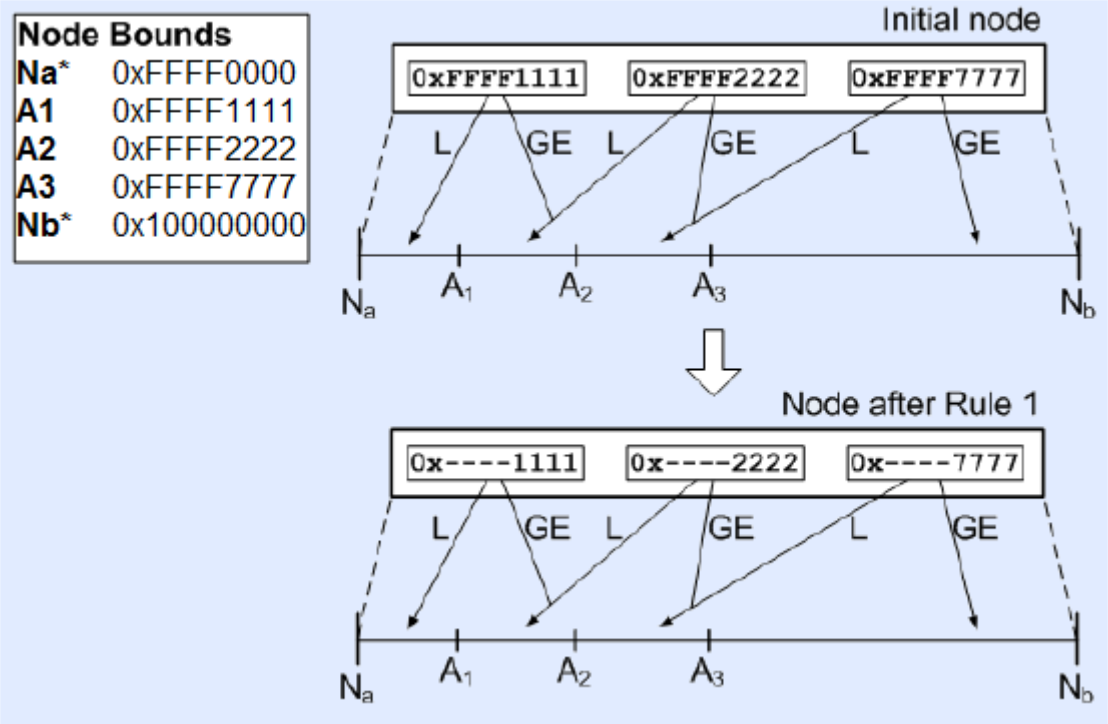

Figure 2.15: Rule 1 – Omit Common Prefix

2.2.1.2. Rule 2 – Share addresses' common prefix

The concept of rule 2 is similar to that of rule 1 but it is limited to the compare bounds only and not the total interval. Specifically, if all the compare bounds share a common prefix (CP) but not the node bounds too (N_a and N_b in rule 1), then it is still possible to perform an optimization.

Specifically, we can compare the corresponding part of the incoming address (AP) with the common prefix part only once, and then we extract the following (where k the number of bounds contained in the node):

- If AP < CP then the correct interval is the [N_a, A_1)
- If AP > CP then the correct interval is the [A_k, N_b)
- If AP = CP then comparing the rest of the bits of the incoming address to the rest of the bits of the address bounds will give the correct interval. The processing is equivalent to the one indicated in the bottom part of figure 2.15 for rule 1

Example of application of rule 2 to a node can be seen in figure 2.16 (source [4]). Applying rule 1 in this case would not be efficient, since the common prefix of the node, corresponding to the interval [DDDD0000, FFFF1110] would be only the two most significant bits. On the contrary, the compare

addresses share the EEEExxxx part. This means that the result can be determined by comparing once the common prefix to the corresponding part of the incoming address (16 bit comparison) and if equal comparing the remaining bits of the compare addresses to the corresponding part of the incoming address (again 16 bit comparison).

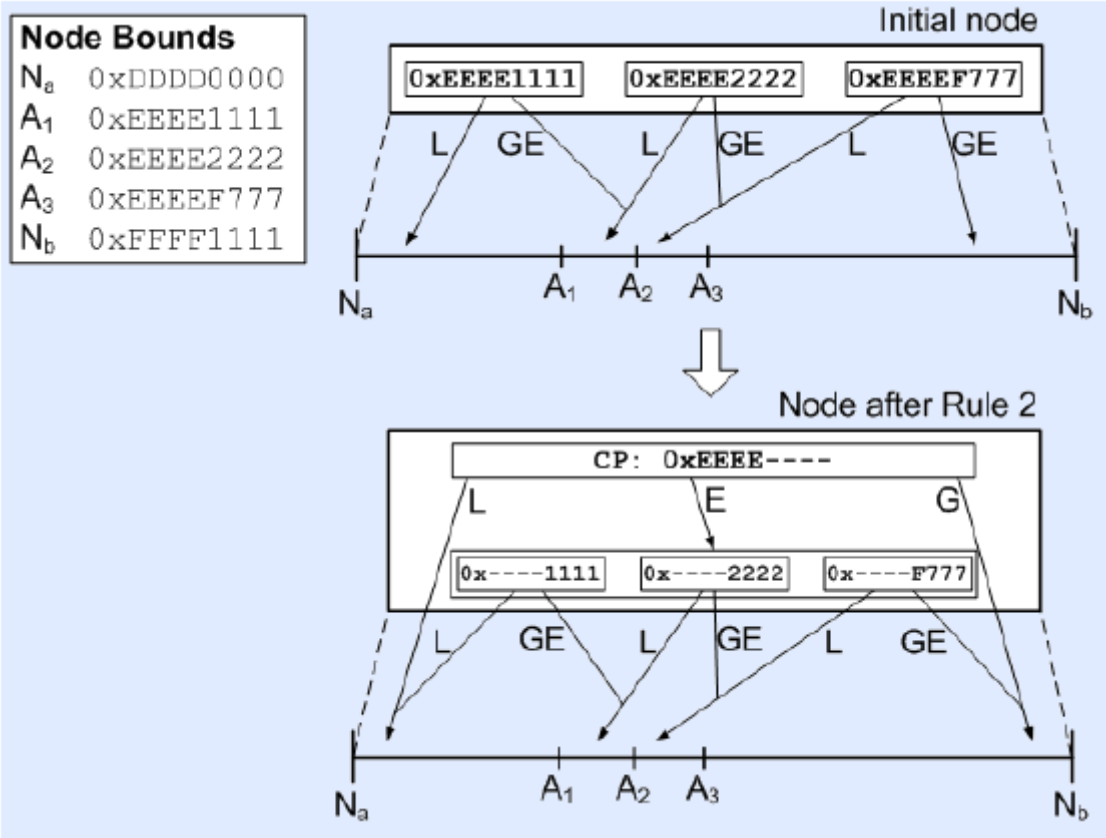

Figure 2.16: Rule 2 – Share Common Prefix

This way we gain in comparator use (fewer bits compared totally) and memory efficiency (initially we had three 32-bits long comparison values, now we have 16 bits prefix and three 16-bits long comparison values). We described the process as sequential, but in reality comparing with the prefix and with the remaining addresses can be performed in parallel.

2.2.1.3. Rule 3 – Share addresses' common suffix

Equivalently to rule 2, rule 3 states that if there is common suffix (CS) this can be shared among the compare addresses. The process of determining the correct interval is the inverse of the process followed when rule 2 is applied. Specifically, initially the non-common suffix part of the compare addresses is compared to the corresponding part of the incoming address. If the result of those comparisons is less or greater but not equal, then the intervals are determined as if we had compared the full addresses. However, if the result of the comparison is equal to some of the (partial) compared

addresses, we need to compare the remaining part of the incoming address (RA) to the common suffix in order to determine the result. Specifically, if A_i is the compare address for which the partial comparison indicated equality, where i in [1, k], then:

- If RA < CS then the correct interval is the $[A_{k-1}, A_k)$
- If RA ≥ CS then the correct interval is the $[A_k, A_{k+1})$

Where we consider that $A_0 \equiv N_a$ and $A_{k+1} \equiv N_b$.

Example of application of rule 3 to a node can be seen in figure 2.16 (source [4]). We notice that the two compare addresses share the xxxx1111 part, thus the parts BBBBxxxx and CCCCxxx are compared to the respective part of the incoming address and if one of these comparisons shows equality the common suffix part needs to be compared to determine the final result. Again, like in the case of rule 2, even though we described the two comparisons as sequential, they can be performed in parallel.

We have to note here that the case of rule 3 to be applicable is rather rare. Its usefulness appears actually only in the higher levels as we will explain in section 3.2.3.

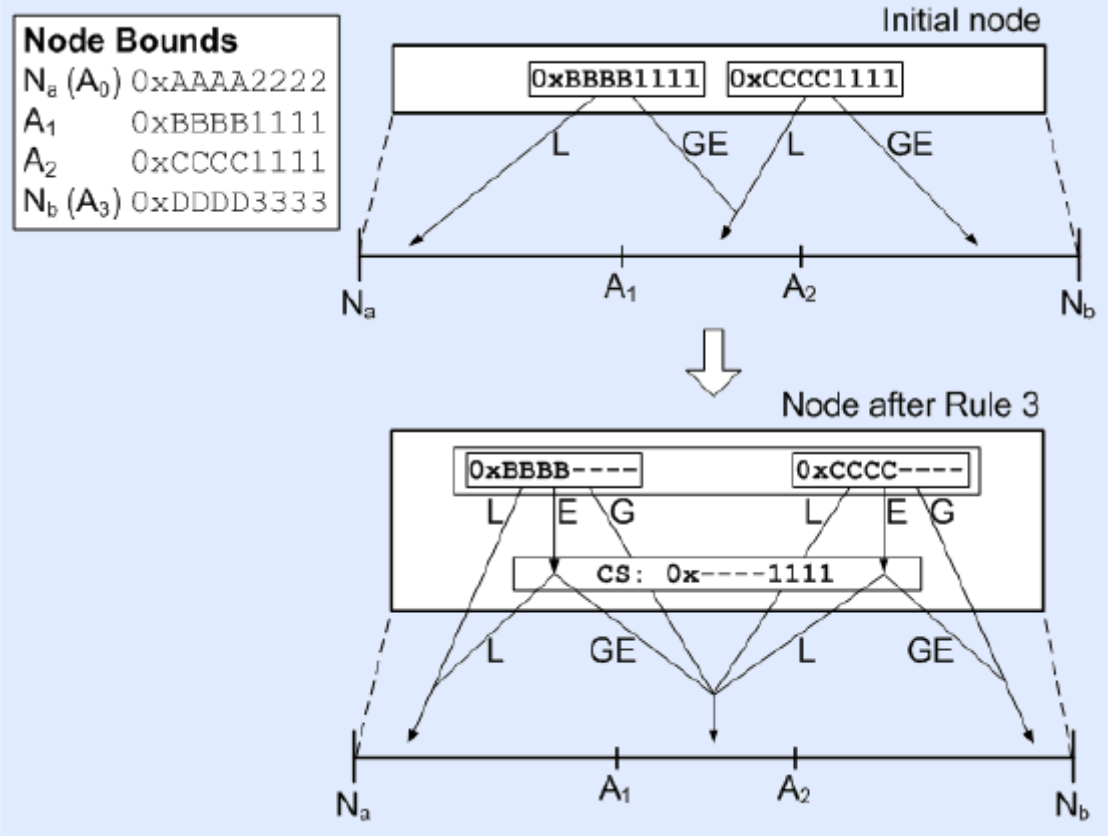

Figure 2.17: Rule 3 – Share Common Suffix

2.2.1.4. Rule 4 – Omit addresses zeroes suffix

Rule 4 probably constitutes the simplest optimization conceptually. It simply states that if all the compare addresses in a node share a zeroes suffix part, this part can be omitted. Comparing only the higher part is equivalent to comparing the full addresses and the zeroes suffix is simply ignored.

In reality, the cases in which rule 4 is applicable is a subset of the cases rule 3 is applicable, since rule 4 demands not only a common suffix part as rule 3 does, but also this common suffix to be zero. However, rule 4 is much simpler and requires fewer operations that rule 3, and for this reason it has been introduced as a separate rule. Moreover, in certain cases (see VLC later), rule 4 can be reapplied separately per bound, thus it is not necessary all the bounds to share the same zero suffix.

Figure 2.17 is indicative of this notion. If it were A_1 = BBBB0000 and A_2 = CCCC0000, then the common suffix would be xxxx0000. However, comparing an unsigned value to zero gives always greater or equal as result, thus the processing is reduced to only comparing the higher part. Figure 2.18 represents the result of application of rule 4 on the node.

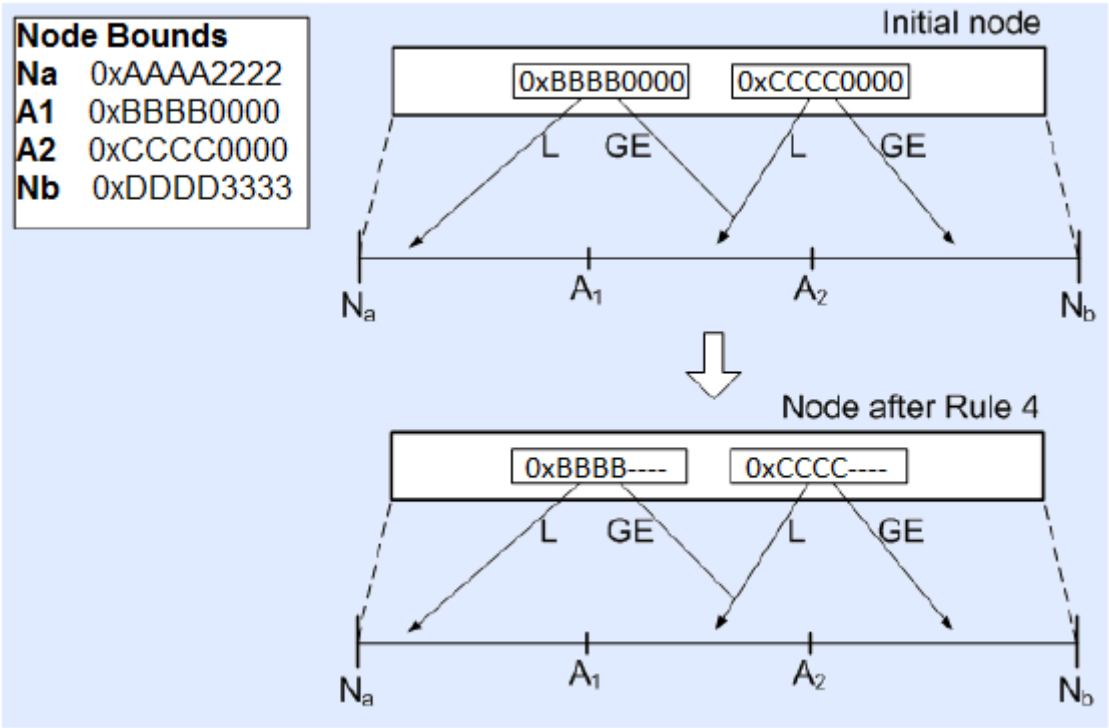

Figure 2.18: Rule 4 – Omit Zero Suffix

2.2.1.5. Rule 5 – Address alignment

Rule 5 is based on the principle stating that for a node with address space [N_a, N_b) performing a comparison between an incoming address A and a comparison value A_i is equivalent with performing a comparison between an incoming address A-N_a and a comparison value A_i-N_a. This offset modifies

the incoming addresses and the comparison values by mapping them to the interval [0, N_a-N_b). Considering that the address range of the node is small this creates a long (zero) common prefix for the modified incoming addresses which can be omitted according to rule 1 (see section 2.2.1.1). Specifically, the length of this common (zero) prefix can be calculated to be equal to W-$ceil(log_2(N_b$-$N_a))$ where W is the address width and $ceil()$ is the round-up function, so we need to preform comparison only for the remaining $ceil(log_2(N_b$-$N_a))$ bits part. Moreover, this subtraction can be also compressed, since we already know that the high common part with be zeroes. Thus, we need to subtract only the last $ceil(log_2(N_b$-$N_a))$ bits of N_a from the incoming address, which are actually the ones we use for comparison.

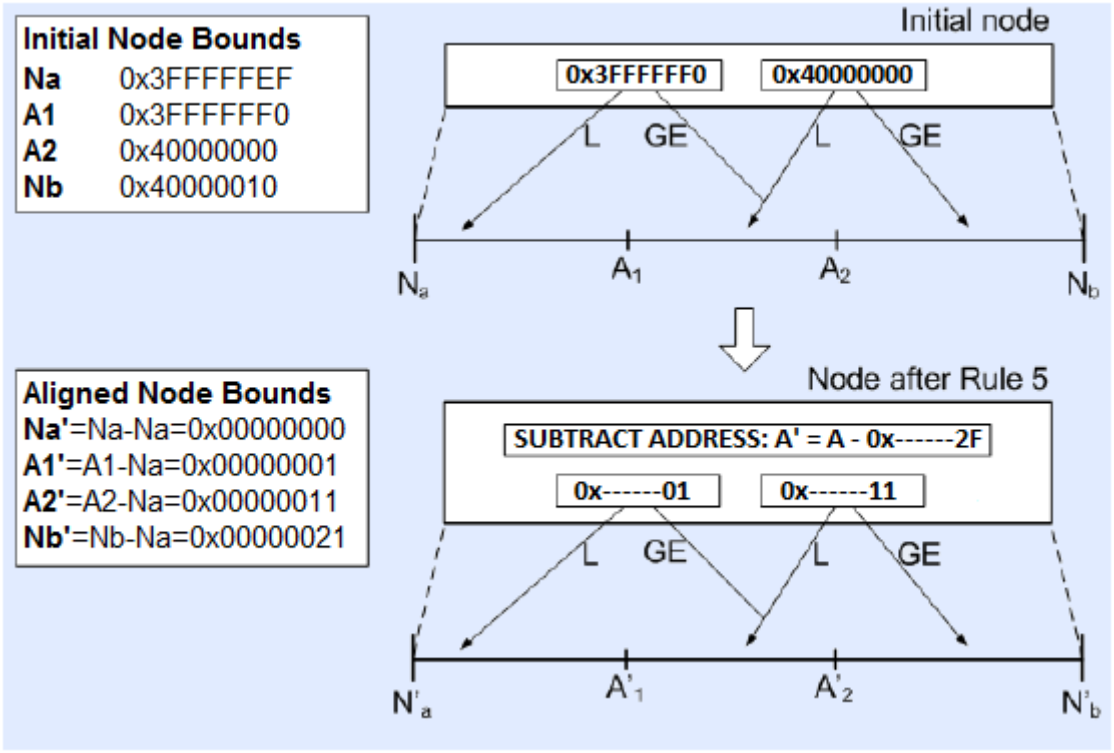

Figure 2.19: Rule 5 – Address Alignment

Figure 2.19 shows an example application of rule 5 on a suitable node. Before the subtraction the common prefix for the specific compare addresses is one bit only. After the subtraction 26 bits are common zeroes and rule 1 can be applied. Note that the subtract value used, 2F, represents the last 6 bits of N_a=3FFFFFFEF. Moreover, figure 2.19 is a nice example of a practical application of rule 3 (considering that rule 5 is applied before rule 3 or 4, but in practice it is more efficient to apply first rules 3 or 4 first to avoid exactly this procedure). Considering the initial node bounds, 3FFFFFF0 and 40000000 the common suffix is also a zero suffix, so rule 4 would be used as more preferable. However, after the application of the subtraction according to rule 5 the new (aligned) node bounds are 00000001 and 00000011, or ------01 and ------11 after the application of rule 1. These new bounds have no zero suffixes; have though a 4-bit long common suffix, allowing the application of rule 3.

From the above example it has been made obvious that rules 1-4 can be applied independently from each other, though rule 2 is a generic version of rule 1, and same applies to rule 3 compared to rule 4. For this reason if both the generic and more specific rules are to be applied the more specific have to be applied first. Rule 5 modifies the results for rules 1 and 2, and thus it has to be applied before them, since rule 5 creates common prefix which can be of different length compared to the initial common prefix. Rules 3 and 4 which refer to the common suffix are not affected, but algorithmically it is preferable to calculate the common (zero) suffix first so that the subtraction is applied only on the variable part of the bounds only, which is very convenient since the subtractor width is limited.

2.2.2. Range Trie Structure

The range trie structure is the result of attempting to apply all of the optimizations corresponding to above mentioned rules on a range tree-like structure. At each node the attempt to apply each of the rules we presented in section 2.2.2 and its subsections is met with various degrees of success, dependent on the bounds which the range trie construction algorithm tries to assimilate into each node. Thus certain rules for each node are applied, and again each applied rule with its own characteristics. The various algorithms of the range trie nodes generation are presented in section 2.2.4.

As a result of these algorithms the constructed nodes contain a fixed size -equal to the memory bandwidth- line with the comparison values. Due to the optimizations, these comparison values are a combination of the following elements: full addresses, partial addresses, common prefix, common suffix. Thus, each node has a different number of comparison values k, which means that it splits the node address range into k+1 intervals (typically at least, there is an exception when N_a=A_1). Moreover, each node contains additional information like the number of comparisons k (implicitly), how are the comparators split to use, which part of the incoming address to compare, the location of the children nodes, if there is common prefix and/or suffix to share/omit and their position (which part of the address they represent), and the subtract value.

If the node is a leaf node, these k+1 intervals called *basic ranges* are defined by bounds created by the initial prefixess and there are no other bounds created by prefixes in them. Thus, there is a unique longest prefix which matches any address in these intervals. Otherwise, if the node is an internal node, meaning that it has other children nodes, the correct branching to the next level node N_i, where i in [1,k+1], must be determined by these k comparisons C_j where j in [1,k]. The process of traversing the range trie when a destination address is required to be looked up is the following.

The range trie traversal during an address lookup always commences from the root node of the structure which is unique and covers the whole address space. Subsequently, in each node visited, the following steps are performed:

I. If rule 5 has been applied and an address alignment has been performed, the subtraction value is subtracted from the indicated (by the common prefix length field) bits of the destination address A.
II. The k comparisons are performed between the indicated (again by the common prefix length field) bits of the modified by step I destination address and the k comparison values. The result of these comparisons is less (L) or greater-or-equal (GE). According to the results of these comparisons:

- If C_1 = L, then N_1 is the correct next-level node to visit
- If C_k = GE, then N_{k+1} is the correct next-level node to visit
- If C_j = GE and C_{j+1} = L where j in [1,k-1], then N_{j+1} is the correct next-level node to visit

III. If there is common prefix sharing, then the corresponding bits of the destination address A are compared with the common prefix. According to the result:
- If result is L, then N_1 is the correct next-level node to visit
- If result is G, then N_{k+1} is the correct next-level node to visit
- If result is E, then the result is determined by the k comparisons in step 2

IV. If there is common suffix sharing, then then the corresponding bits of the destination address A are compared with the common suffix. According to the result:
- If result is GE, then the result is the one obtained by step 2, meaning N_i.
- If result is L, then the result is the previous node of the one obtained by step 2, meaning N_{i-1}. Exception appears when i=1, in which case N_i is the correct result

The processing was presented sequentially, but the comparisons involved in steps II-IV can be performed in parallel in order to minimize the execution time, the final result requires only inspecting the results of all the comparators also in parallel.

Steps I-IV are repeated when visiting each node until a leaf node is visited. The last iteration of steps I-IV then provides as output an address range which can be matched uniquely to a longest prefix.

At this point we will refer an implementation-related detail. In practice, for reasons which are explained in detail in [4] and [5], the nodes for each level are stored sequentially into the node memories. We will briefly refer here the logic behind this decision. The initial reason is that in order to access a variable number of children nodes per parent node we need some mechanism to point to the children nodes. Since we want to minimize the data stored, we simply store a single pointer for the first child of a node and the rest of the children are stored sequentially into the next memory locations. However, as it can be seen in figure 2.14, a node may have two parents (unless a node splitting has been performed, see [8]), which forces the pointer of the second parent node in a specific position determined by the pointer of the first parent node and the number of children of the first parent. For this reason the simplest solution is to place the nodes of the whole level sequentially. This limits the flexibility of the design (static range trie) but allows better memory utilization than the one achieved by node splitting (see [8]).

2.2.3. Range Trie Construction

In section 2.2.1 the rules/optimizations which constitute the core idea behind the range trie scheme were explained. Section 2.2.2 presented the methodology to combine all these rules leading to conceptually solid algorithm. This includes the data stored in the nodes and the way they are used in order to ensure correctness. The algorithm describes a) how a given node is processed in order to determine the correct branching, and based on this b) the method of traversing the range trie until the correct range is retrieved. However, no information has been given yet concerning the method of creating the nodes composing the range trie.

Ioannis Sourdis et. al. in [6] proposed an automated process of nodes generation. Due to the complexity of the range trie algorithm and the size of the problem (the size of the problem is the size of the lookup table measured as the number of distinct bounds contained in it), exhaustive search of the solutions space have been considered as infeasible. For this reason heuristic methods have been utilized. Considering the size of the problem these can be fast and offer high quality, though not necessarily optimal, solutions.

Optimization criteria are the depth of the range trie and the memory required. Consequently, the heuristics have to create balanced range trie structures and assimilate as many bounds as possible per node. Two approaches have been tested: *top-down* (meaning that the root node is constructed first and the processing continues with nodes in progressively lower levels) and *bottom-up* (initially nodes based on the prefix bounds are constructed and the processing continues to higher level nodes progressively until a single root node has been constructed). Each approach has been implemented in two versions: single length comparisons or variable length comparisons within the nodes, see [6] and [7] for more details. We will present here a brief description of those:

- Top-Down with Single Length Comparisons (TD-SLC):
  I. Apply rules 5, 1, 4: address alignment, omit the common prefix of the node, omit the addresses zero suffix.
  II. Determine the comparison width that maximizes the number of branches.
  III. Omit for all the addresses the suffixes that exceed the comparison width by assuming that they are equal to zero.
  IV. Create the defined groups.
  V. Merge adjacent groups until the number of needed comparisons is reduced to the number of available comparison resources. Apply range trie rules 2 and 3 to share common address prefixes and suffixes. The resulting groups are the node branches and their borders are the comparisons to perform.
  VI. Apply Steps I-V recursively for all the children nodes.
  VII. Terminate when each group contains only a unique address range (defined by the prefix bounds)
- Top-Down with Variable Length Comparisons (TD-VLC):
  - Same as TD-SLC, but Step V is modified: Merge adjacent small groups and split large groups. Splitting is achieved by adding an extra comparison of longer length.
- Bottom-Up with Single Length Comparisons (BU-SLC):
  I. Apply the Range Trie rules and select the first b addresses $A_i \geq G_{down}$ (A_i in [A_j, A_{j+1}, ..., A_{j+b}]) that can be compared at one node based on the single-length available comparison resources. G_{down} is initially 0.
  II. Set as the selected group's upper bound G_{up} the address in the address range (A_{j+t}, A_{j+b}] with the longest zero-value suffix, where $t/b = C$ defined by the user. The resulting group maps to the node with range [G_{down}, G_{up}).
  III. Repeat Step II by setting $G_{down\text{-}new} = G_{up\text{-}old}$. Step III repeats itself until all bounds A_i have been grouped.
  IV. Repeat Steps I-III recursively by considering that the new A_i' set is the old G_{up} set. Note thus that G_{up} has to be stored for use for the next level nodes.
  V. Terminate when the whole new set of A_i' is processed in a single iteration creating a root node.
- Bottom-Up with Variable Length Comparisons (BU-VLC):

- This is same as BU-SLC, except Step 1 where the available comparison resources may now be of variable lengths.

Out of these methods the two bottom-up ones construct the most efficient range tries; however, they require more processing time than the top-down ones (see [7]).

2.2.4. Range Trie Support for Longest Prefix Match

Until now we described how the range trie determines in which of the basic address ranges defined by the prefix bounds a given destination address belongs to. However, we need a mechanism to map this basic address range to the prefix and its actions. The additional functionality added to the range trie structure in order to perform the longest prefix match will be presented in this section.

To provide the desired functionality, the range trie has to embed in its structure prefix information. Following the same methodology as in range trees, see [18], the prefix information is stored in the intermediate nodes too and not only at the leaves. However, the limited memory bandwidth forces us to minimize the data we store on the nodes. For this reason the actual data corresponding to the address lookup (actions/prefixes) are stored in a different (external) array called action table.

An important observation here is that the leaf nodes contain multiple intervals, all of which need to be mapped to some prefix (if they have not been mapped by their parent nodes already), and there are no further nodes to contain these prefix information. As a result, the nodes store information not about the address range of the node, but about each of the intervals its bounds are defining. Considering that the number of intervals per node is variable, this complicates the whole scheme.

According to the above, the data which are necessary to be stored in the nodes in order to enable the longest prefix match functionality are:

- The prefix lengths of the longest prefix covering each interval defined in the node. If no prefix covers that interval then this is indicated as invalid prefix length. If the longest matching prefix already covers that interval in some parent node then it is not needed to be stored again in the child node.
- The action pointers to the prefix/action. This action pointer is necessary in order to access the external array containing the prefixes/actions. Again, if the prefix (including its actions) is stored at some of the parent nodes then the action pointer is declared as invalid. Same happens if there is no corresponding prefix for that interval.

Explanation about how exactly this works in the hardware implementation can be seen in [4], [8] and [5]. We will just refer here that the software indicates the area where the action will be located through the node pointer. The exact location of the action in the action table is though determined by the hardware. If p is the node pointer and $k+1$ the number of intervals the node defines through its bounds then the hardware selects an address i between p and $p+k$. The action pointer is then defined by the hardware as $i-p$. The parts related to the software will be also presented in section 4.4.2. However, for clarification reasons we will note at this point that the process of obtaining the correct prefix requires us to search the range trie up to the leaf nodes in case a longer matching prefix is stored there. The one selected as output is the last matching prefix detected. The range trie has been implemented in such a way that no backtracking is required.

2.2.5. Dynamic Range Tries

The range trie structure as defined until now is a highly optimized version of the range trees. These optimizations offer superior scaling characteristics in terms of performance and memory requirements as we see later in section 3.2, however, they also increase the complexity of the whole scheme. As we described in section 2.2.3, the construction of the range trie is a complex process based on heuristic algorithms. This fact makes range tries inflexible; any lookup scheme should have a mechanism of updating itself when the respective lookup table is updated, but once created, it is impossible to add new prefixes (with bounds defining their range) into the range trie with the same degree of efficiency as performed during its initial construction. The reason is that the heuristics already try to make full use of the available bandwidth in order to reduce memory requirements and range trie levels, thus the existing nodes can support limited or not at all new bounds and in the general case new nodes have to be created. Adding a node in the existing levels would be a very complex and mainly time consuming process. Characteristically we will refer that such a new node not only would require changes in the pointers of all the subsequent nodes of the same level, but possibly also changes in the higher levels too. For this reason the new node has to be created in lower levels, taking the place of a leaf of the original range trie. This however leads to increase in the number of levels and also drop of quality of the range trie structure.

For the above reasons, and taking into consideration that the range trie targets hardware implementation in order to reveal its potential, a new additional structure has been invented. The hardware supports the range trie structure created by the heuristics as defined until now (called *fixed levels part* from now on), but it also contains a number of additional levels where *new nodes can be created dynamically* when new bounds are inserted, called *spare levels part*. In contrast to the fixed levels part, nodes and their position are defined by the hardware autonomously (no software support) and due to this the nodes are much simpler compared to the fixed part nodes. Specifically, the nodes the hardware creates constitute in reality a *dynamic range tree* structure (see section 2.1.8), where comparisons are performed between the destination address and the *full* bounds without further complications.

The reason behind this decision is that the hardware should be able to handle updates fast without having to wait for the software to collect all the necessary data, process them to create the new node based on the range trie rules and finally send it back, which would result in drop in performance. This would also increase the complexity of both hardware and software significantly and overall would offer only a small improvement in quality. On the other hand, as we will discuss also in chapter 4, the quality of the spare level structure can be improved when required thanks to the software support for rebalancing the range trees (see sections 4.4.3 and 4.4.4), while the major improvement in quality is achieved by recreating the whole range trie structure in software and passing it anew in hardware by integrating also the new bounds located in the spare levels (see sections 4.4.2, 4.4.3 and 4.4.5).

Figure 2.20 contains the schematic of a dynamic range trie including both the fixed levels part and the spare levels part.

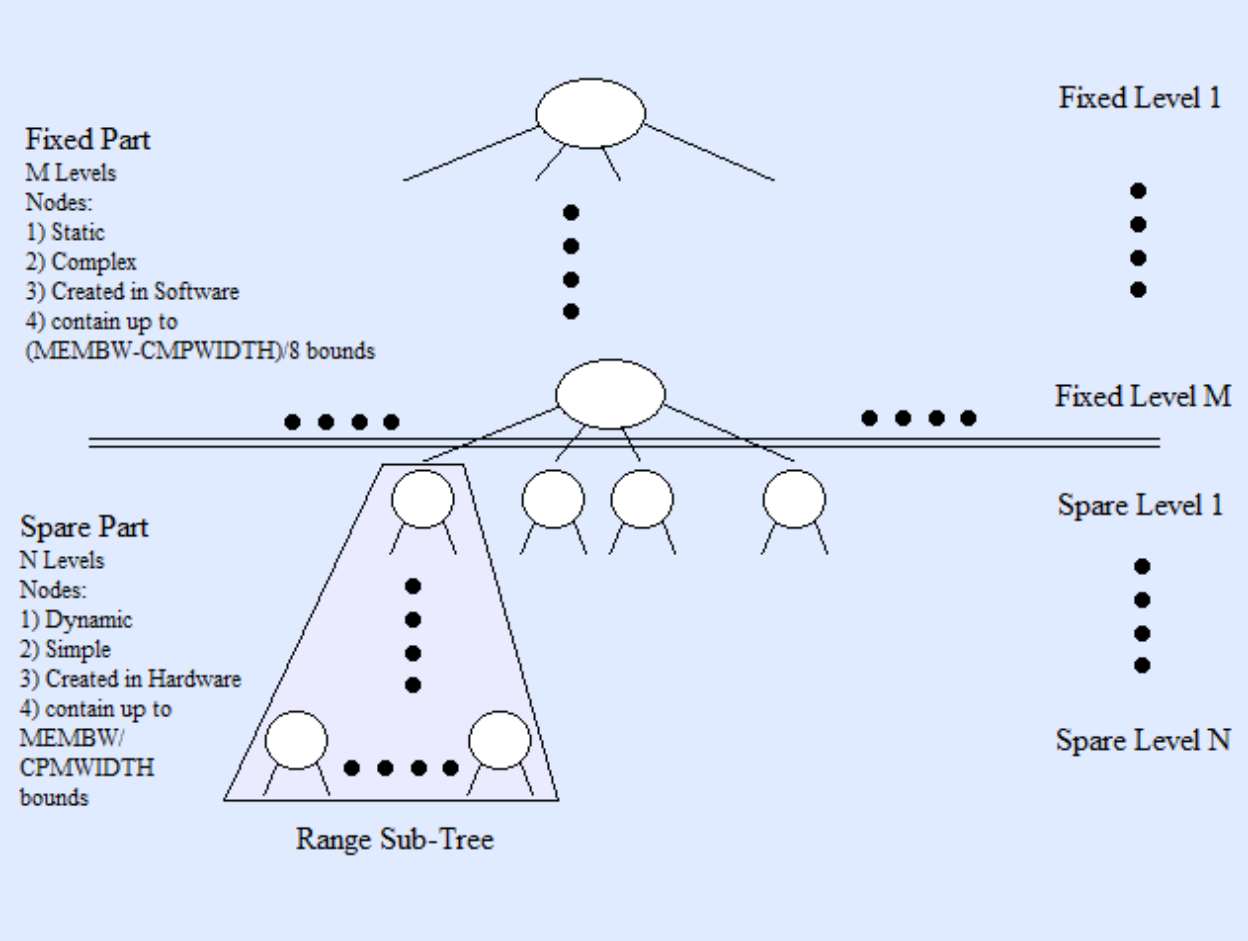

Figure 2.20: Dynamic Range Trie

2.3. Summary

In section 2.2 we presented in detail the concept of the range tries. The basic ideas, the method of combining them into an optimization algorithm, the conceptual and structural design of the range trie along with some brief implementation related information constitute the basis on which we will develop the work presented in this thesis (see chapters 2 and 3) and the final conclusions (presented in chapter 5). Relevant algorithmic address lookup schemes have been presented in section 2.1, allowing us a theoretical comparison with the range trie scheme. From this comparison it has become apparent that the range trie scheme has the potential to accumulate a desirable set of properties in higher degree than other competing techniques:

- Low latency since the produced structure has low depth compare to other schemes and considering that the extra processing cost is performed in comparable or even higher speed as simpler solutions.
- High Throughput comparable to other schemes. No characteristics of range tries provide any advantages or disadvantages. Pipelining (see [5] and [8]) and parallelization can increase it substantially.

- Low Memory Requirements since nodes have higher branching factor compared to the used bandwidth of other relevant techniques.
- Good Scalability since it combines the good lookup-table-size scalability of tries with the good address-length scalability of the range trees.

The above characteristics are the main motivation to further explore the possibilities of the range trie scheme.

There are several details still missing from a complete description of the range trie scheme. The reason for this is that they are implementation-related and not in the core concept of the range trie design. The hardware implementations can be found in [4], [5] and [8]. The software has to be compatible with the specific hardware implementations and for this reason relevant details will be presented as introduction to chapter 3 (see section 3.1). In [7] and [8] an initial version of the configuration generation software has been created, and the work related to chapter 3 is a continuation and refinement of this software.

3 Range Trie Configuration Generation Software

As we mentioned in section 2.2.3, the optimal node construction for the range trie structure is a complex procedure if we take into account the size of the problem and the number of optimizations that have to be performed. For this reason the respective procedure is based on heuristic methods. The aim of these heuristics is to create a high quality range trie structure matching the set of prefixes contained in the lookup table.

The base of the range trie structure as described in section 2.2 is similar to that of a range tree from the perspective that comparisons have to be performed between the incoming (parts of) addresses and the respective (parts of) address ranges bounds (referred as bounds for simplicity reasons). Thus, starting point of any relevant heuristic is the set of bounds defined by the prefixes contained in the lookup table.

As we mentioned in section 2.2.3, there are two approaches to construct the range trie nodes. The first is the top-down, starting from the root and proceeding with the lower levels until eventually the prefix-bounds are used in the leaf nodes of the range trie. The second is the bottom-up, which starts from the prefix-bounds and the leaf nodes of the range trie, and builds subsequently the higher level nodes up to the root node.

According to the work presented in [7], the bottom-up approach is slower than the top-down one, but produces much higher quality range tries. Specifically, bottom-up approach creates balanced range trie structures, containing less nodes with higher average branching factor, leading to lower memory requirements, better resource utilization, and shorter tries which reduces the address lookup latency. These characteristics constitute the main objectives of the range trie. For this reason the bottom-up approach is favorable and thus the work related to this thesis is based on this.

Another distinction is the single-length or variable-length comparisons per node. According to [7], by using variable-length comparisons per node it is possible to achieve higher branching factor per node, at the cost of higher complexity and higher memory requirements in order to store the comparator modes. According to the encoding scheme proposed in [7] and implemented in [4] for 32-bit comparators 3 bits are used for the comparator mode with variable-length comparisons compared to 2 for single-length comparators. Equivalently for 128-bit comparators 15 bits are needed for variable-length comparators in contrast to 3 bits for single-length. Focus has been also given on this issue in [7], but we further analyzed and expanded these schemes as we will discuss in the following sections of this chapter.

For the case of the bottom-up construction of the range trie an initial version of the range trie configuration generation software was created as part of [7], while some modifications were applied on it in relation to [8], were the main focus though was the hardware. Until the present thesis though and the work presented in [5], no working hardware implementation of the range trie had been ported on actual hardware (FPGA). Beyond the work presented in [5] which refers to the hardware part of

the range trie, the work related to this thesis was also essential since it also takes into account hardware-related details in order to provide portable to the hardware range tries.

Our purpose was to create a range trie configuration generation software version which will exhibit the following characteristics:

- Correctness: The results must be a valid configuration for the hardware implementation of the range trie for any given lookup table. This refers both to the format of the output file and also in the content.
- Reliability: The software must be stable, avoiding crashes and memory leaks.
- Versatility: The software must be able to create the appropriate range trie configurations for various hardware implementations. Beyond some basic characteristics which are common (correctness has to be ensured for these), the hardware implementations may vary significantly, indicating that highly parametric software is necessary.
- Flexibility: Even for a given hardware implementation, there are various different range tries which can be constructed. It is desirable to be able to alter the algorithm effectively, mainly for testing purposes, but also for tuning to the specific lookup table.
- Performance: Low execution time for our software is desirable but not the most important characteristic. Of higher importance is the ability of our software to handle large lookup tables, since the algorithm is highly memory demanding.
- Quality: The resulting range tries must be balanced, short, and containing the least possible number of nodes.

In the rest of this chapter we will present how we met these criteria, accompanied by the respective results. Specifically, in section 3.1 we will present the modifications and additions we performed in order to improve the software. In section 3.2 we will present various testing results based on the final version of the software. Finally, in section 3.3 we will conclude this chapter by analyzing the results and making some interesting observations.

3.1. Software Modifications and Additions

As we mentioned earlier, the construction approach our software follows is bottom-up, and can accommodate both single length and variable length comparisons (BU-SLC and BU-VLC as they are referred in [7]).

A design decision we made was to follow the CMPN mode, which means that the *address width* and the *comparator width* match. The alternative mode is called CMP32, in which the comparators are of fixed length 32-bits independent of the address width. This however leads to severe complications in the software and most importantly to poor results (exception to this is the case the bounds follow a Gaussian distribution in the address space, in which the results improve [7]). Note at this point that *comparator width* should not be confused with *comparison width.* One single comparator can accommodate one equal-width comparison or split in multiple lower-width comparisons. To explain

briefly the reason behind the poor results and also the software complication we should note than in CMPN mode, no matter how "inconvenient" a bound is (low common prefix/suffix length), we can always use a full comparator in order to compare it to the incoming address, and possibly having more spare bandwidth in the same node in order to perform more bound comparisons. On the other hand, in CMP32 mode, if the bound is "inconvenient" we may need multiple nodes in order to perform just one bound comparison, which leads to waste of memory and possibly to increase in the range trie levels.

Beyond this design decision, the logic behind the range trie configuration generation software is not different from [7]. The same rules are applied as in [7] and in similar order (see section 3.1.2 for some changes). The core element of the software is a function which accepts as input a set of bounds in increasing order and the high bound of the previous node of that level. On those it applies the optimizations presented in section 2.2 as rules 1-5 in order to create a new node. Aim of this function is, through these optimizations, to "pack" as many bounds as possible in the same node and at the same time to create an upper node-bound which will allow easier processing of the higher level nodes. The components of this function in order of usage in it are the following, considering that they are applied (multiple times) on a sorted array of bounds.

- Common (Zero) Suffix Calculation: The common (zero) suffix is calculated so that we can determine which part of the least significant bits of the considered bounds can be shared (omitted).

- Address Alignment: This component determines if rule 5 should be applied based on the current set of considered bounds. Specifically, ignoring the common (zero) suffix, the low node-bound is subtracted from the last bound of the considered set, and if the result fits in a number of bits equal to the subtractor width then rule 5 is applicable, otherwise not.

- Common Prefix Calculation: The common prefix is calculated for the bounds only (rule 2: share common prefix) or for the bounds and the node bounds (rule 1: omit common prefix).

- Determine Comparisons Width: For single-length comparisons this is *rtupot(N-CP-CS)* where *N* is the address length, *CP* is the currently computed common prefix, *CS* is the currently computed common suffix, while *rtupot()* is a function which *rounds to the upper (integer) power of two* (example 7 is rounded to 8, 13 is rounded to 16 and 17 is rounded at 32). For variable-length comparisons, comparison widths is attempted to be reduced even further by omitting any additional zero suffix per individual bound (after the address alignment, preserving the same common prefix length and again rounding the width to the upper power of two).

- Validation: This step is applicable only for variable-length comparisons per node. The problem is that the comparators cannot split in any possible combination adding up to their width (for example for 32 bits comparator the combination 8-16-8 is invalid). For this reason an algorithm is invoked which expands the values of the comparisons wherever needed in order to reach a valid series of comparisons.

The above steps are performed repeatedly per node until the optimal valid node configuration has been determined (common prefix, common suffix, subtract value, bounds and their respective comparison widths). Note that optimality in this case refers to the number of bounds in the node and

not to the optimal range trie configuration in general. In [7] this procedure is called *Group Bounds*, but in reality it consists of iterative application of the above steps. The next steps are:

- Upper Bound Selection: The upper node bound has to be selected with an effort to maximize the final number of bounds in the node and achieve an as long as possible zero suffix for the upper bound itself. The latter is desirable in order to create "convenient" bounds for the higher level nodes. These criteria are conflicting since the more bounds we want to integrate into the node (from the group created from the previous steps), the narrower the interval for upper bound selection and thus less probable to obtain a very long zero suffix. The simple method to achieve that is based on a variable *MINBORDERS* which represents the minimum *fill factor f* (referring to the bounds group size). This defines the area from which the address with the longest zero suffix is selected as upper bound. A more sophisticated method uses an optimization criterion in that area instead of selecting the longest-zero-suffix address. This has been nicely implemented in [7] and thus we preserved it in our code.

In [7] an additional step is described, called *Reduce Bits*, followed by a validation step, but we integrated it in the first steps for efficiency reasons. With the above steps the node has been constructed. The process continues with the rest of the nodes, both in the same level for the remaining bounds and in higher levels for their parent nodes, until the root node is created. The last step is called:

- Output: The set of the created nodes has to be printed into a configuration format defined by us. This is a generic parametric format, compatible with any appropriate hardware implementation of the range trie (see [8], there have been major format changes from the [7] version). Specific hardware implementation details must be known to the software since they are used as parameters in order to produce a matching configuration file (see section 3.1.2 for more details). The implemented software for this thesis contains additional logic into this module since the compressed bounds for comparisons are created and printed simultaneously in the output module, based on the uncompressed bounds stored at the nodes. This allows us to store less information in the nodes while preserving the simplicity of the node structure representation. This is important since it allows easy visual inspection of the intermediate results.

The next four sections address different aspects of the range trie configuration generation software. Specifically, in section 3.1.1 we will present the actions we took in order to ensure the correctness and the reliability of the software, in section 3.1.2 we will present the versatility of our software and the various modes of operation (flexibility), in section 3.1.3 we will discuss the methodology followed in order to improve the performance and the abilities of our software, while section 3.1.4 we will discuss about the quality of the processing in regard to the output results.

3.1.1. Software Correctness and Reliability

The first element of the software is the one that obtains the input lookup table based on which the algorithm creates the range trie. The initial software contained a method to read a lookup table from file which was adequate and thus has not been modified. For our testing purposes our software makes heavy use of randomly generated lookup tables. The initial software contained such a generator for IPv4 addresses. The IPv6 version also contained such a generator but it appeared to be unreliable (we

observed crashes due to segmentation faults). For this reason we decided to create our own random prefix generator which is generic as far as the maximum prefix length and the number of prefixes are concerned. It provides standard functionality like uniform prefix/prefix-length generation, checking if the generated prefixes are unique, storing them if asked in memory or in files in the hard disk, and lastly creating, sorting and removing duplicate bounds from the resulting prefix-bounds array.

The resulting lookup tables are accurate (no inconsistencies), but the check for unique prefixes and generation of replacing prefixes when duplicates are detected is slow (complexity $O(n^2)$ in our code). For this reason we loosened the rule for unique prefixes and performed only a uniqueness check on the sorted array of final bounds ($O(n)$ complexity). To reduce the probability of multiple instances of the same prefix we assigned much greater weight to longer prefixes because the probability of duplication decreases exponentially to the prefix length. This results in total complexity defined by the function which sorts the final array with the bounds, and for good sorting algorithms it has complexity $O(nlogn)$.

Subsequently, a series of bugs were detected and fixed. These belonged in various categories.

- Crashes/Segmentation Faults: Even the initial version of the code was parametric (and the final presented in this thesis is even more generic), while the algorithm itself is heavily data depended. The control flow is complex since there are many different cases and modes of operation. Especially the initial version of the code performed many dynamic memory arrays/structures allocations which results in complex memory management. This, combined with data-dependent arrays/structures indexing, has as a consequence high error probability. We confronted this in several occasions. Beyond the erroneous random IPv6 lookup table generator which produced segmentation faults consistently, many components of the code also produced segmentation faults or infinite loops depending on the data they were handling.

- Wrong Format: In certain nodes the configuration file format was violated. These mistakes were detected in the output printing module but also in the node creation module, since both created data for printing into the configuration file. Due to those, certain parts of the code were revamped completely, while others were analyzed and fixed. We improved the code quality in this point by moving all the configuration file printing logic into the output module. Thus, any format related error can be easily traced back to the output module. This also allows even erroneous range tries to be properly printed, enabling easier inspection and analysis of the results (even if they are wrong we can directly see what is stored in the nodes from the configuration file).

- Erroneous Range Trie: The resulting configuration file may have a valid format, but there may still be various mistakes. Rather than defining any extensive classification for those, which would be pointless, we will simply refer that the result of those is that the configuration file contains the description of a range trie which does not correspond correctly to the initial lookup table. Most of those were *programming errors:*

  - Erroneous node creation process (wrong compressed comparison bounds were detected, wrong common prefix length calculation in edge cases only, mismatch or invalid comparator width in the VLC case, wrong zero suffix length calculation when combined with non-zero subtract value etc.).

  - Erroneous pointers in the output module (mismatch parent-children). The source of those was the node splitting module, a complicated piece of code which splits the nodes top-

down according to the bounds of the parent nodes. One of the previous versions of our code still uses it and the pointers have been corrected according to it, but the final version does not use it thus the pointer assignment has become much simpler.

Fixing these mistakes was the first step to the direction of the creation of our software. The resulting configuration can be verified by inspection of the configuration file, inspection of the intermediate results, or by verification through the process described in section 4.4.2 combined with the work presented in [5].

We have to mention at this point that we did not use some methodology to formally prove the correctness of our software since such methods are very complex and time consuming. However, we followed some general principles which ensure good quality code:

- Functions are created for pieces of code which are needed repeatedly. Since function calls consume time we tried to minimize them by creating functions only when necessary. Overall, a balance had to be maintained between code modularity and performance.

- We gave generic solutions to each algorithmic part. Rather than breaking down our algorithm in individual problems for the various parameters we created in all possible cases a parametric algorithm. The edge cases are handled separately in basic level and not by creating a new complementary algorithm specifically for them. This allows us to have a generic approach into the code which reveals more easily possible bugs.

- Recursive calls were avoided completely in the software (referring to the configuration file generation software, recursion is used in the software presented later in section 4.4.4). This creates a relatively simple control flow and data flow compared to the complexity of the problem.

- Static arrays or algorithmic optimizations which omit certain arrays completely enable easier memory management, leading to fewer bugs.

- Proper variable naming/renaming. Considering that we did not create the whole code from scratch, certain names have been preserved. The majority though of the code has been revamped partially (fixes) or totally (rewrite from scratch). In all cases we gave names to variables signifying their purpose in the code.

- Format, indentation and commenting the code was very handy in avoiding bugs. If we take into account that at start the code was malfunctioning and not understandable directly due to its innate complexity, these elements helped us mainly in analyzing the control/data flow in order to understand the logic/purpose of each piece of code.

Firstly creating a correct and stable algorithmic core of our code enabled us to add additional functionality, improve the performance and produce quality results. These will be discussed in the following sections.

3.1.2. Software Flexibility/Versatility

As we mentioned already, our code exhibits great versatility and flexibility. The versatility refers to the ability of the software to provide correct configurations for a big variety of range trie hardware implementations. The flexibility refers to the ability of the code to provide different range trie structures for the same hardware implementation, each one exhibiting slightly different characteristics in quality (node number, tree levels) and performance (execution time).

We achieved this by applying parametric algorithms whenever possible and by creating structures which enable us to handle easily the control flow of the software, taking advantage of the natural modularity of the code (based on the steps presented earlier in section 3.1).

Specifically, the software can generate range trie configurations for various hardware implementations defined by the following characteristics:

- Address Length: In practice, of interest is the address length (and bounds' length consequently) defined in IPv4 and IPv6, meaning 32 and 128 bits respectively. However, our software is generic meaning that it has the ability to process range tries for any address length equal to an integer power of two (minimum 8 since the comparators cannot split further). Moreover, note that we implemented the CMPN algorithm, meaning that the comparators used in the hardware have the ability if needed to perform a full comparison between a bound and an incoming address.

- Number of Prefixes/Bounds: For obvious reasons the software has to be able to process and construct a range trie for any size of lookup table. See section 3.2 for related results.

- Memory Bandwidth: This in reality represents the available number of bits dedicated to comparison values. Depending on the configuration of the node, these bits represent either only bounds or bounds and the respective common shared prefix/suffix. Note that there is additional memory required for the remaining node-related information in the hardware, but this is not considered to contribute in the memory bandwidth, without meaning that we can ignore it though. This additional memory contains additional information required from the address range lookup process: subtraction value, next level pointer, comparator modes etc. Typical value for the memory bandwidth is 256 bits; however we performed testing with higher values in order to evaluate the performance of our algorithm and the resulting range trie characteristics. A limitation imposed by the range trie structure itself is that it should be:

  $$MEMORY_BANDWIDTH \geq 2 * COMPARATOR_WIDTH$$

  since a full comparator width is utilized when needed by the common prefix/suffix values.

- Node Address Length: This refers to the physical addressing of the hardware memories. Considering that the hardware implementation is fixed (the hardware cannot change in order to accommodate different range tries), the addressing of the nodes should be performed based on the same length provided that the number of nodes is not large enough to cause "address overflow", meaning that a larger number of bits is necessary in order to address the given number of constructed nodes.

- Node Pointer Length: Equivalently to the above case, the pointer length can and should have parametric size.

- Subtractor Width: This hardware imposed restriction should be never ignored by the software. Certain designs may omit some functionality of the range trie like this for simplicity or technology-related reasons.

- Variable/Single-Length Comparisons: The hardware also contains comparators which can be used in different modes of operation. The idea as presented in [7] is that a node can contain comparisons of a single length or variable length. Our code has the ability to operate and produce range tries for both modes. However, we introduced an additional notion: *variable-length per node/single-length per comparator (V/SLC).*

Single-length comparisons per node can lead to great memory waste since an "inconvenient" bound can increase the comparisons width for the whole node. On the other hand, variable-length comparisons mode per node and per comparator is more flexible but requires complex encoding in the software and the hardware, which also requires a significant number of bits. For this reason, during the work for this thesis we invented the variable-length per node/single-length per comparator mode. This scheme achieves low encoding requirements and hardware complexity, equal to the single-length comparisons mode, while limiting slightly the flexibility of the range trie comparisons compared to the full VLC version and having similar software complexity to it.

The complexity of the encoding scheme for variable-length comparisons per comparator can be seen in figure 3.1, where a 128-bit comparator is presented. The comparator can be split as it is shown in the figure to perform independent shorter-width comparisons. The encoding for the 128-bit requires 15 bits and represents the connections which have to be performed in the hardware between the basic 8-bit long comparators in order to create the appropriate comparison widths. This connection structure which resembles a binary tree dictates that when a connection is performed at some higher level, then all the underlying connections are also performed.

To clarify this with a schematic example, let us consider a random valid series of comparison widths like the {8, 8, 16, 32, 64} presented in figure 3.1. This series of comparisons is placed on the tree as indicated by the stars. This means that to achieve the correct comparison widths in the hardware, all the connections under the respective comparison fields have to be performed. This is signified by the encoding created on the software. The bit number (from left to right) for each connection is also indicated in the figure 3.1, and as it is easy to observe it follows a regular pattern. Instead of mapping the connections to the encoding bits one to one, we signify the connections by assigning 1 to the bit of the top connection for a comparison width and all the subsequent connections are implied. In this way we reserve the number "all ones" (111111111111111) to signify that the comparator is not used. For the specific example, the resulting encoding is 010001000000010 since the top connections for the respective series of widths are #14, #6 and #2 (note that the number of ones in the encoding is equal to the number of comparison widths in the set greater than 8).

We have to make three observations at this point. First, there is no respective range trie hardware implementation for IPv6 at the moment, so the encoding can be modified if needed to match the hardware implementation. Second, this encoding scheme represents a generalization (expansion) of the scheme currently used in the IPv4 implementation, the algorithm is parametric to the comparator width and the encoding follows the same logic as presented. Third, personal suggestion for future change is to perform a one-to-one mapping of the encoding bits to the connections. This will not complicate the software further (it is rather complex already) but will greatly simplify the hardware. The unused comparator can be signified with a different encoding, we propose one 1 followed by 0's until the end of the encoding string.

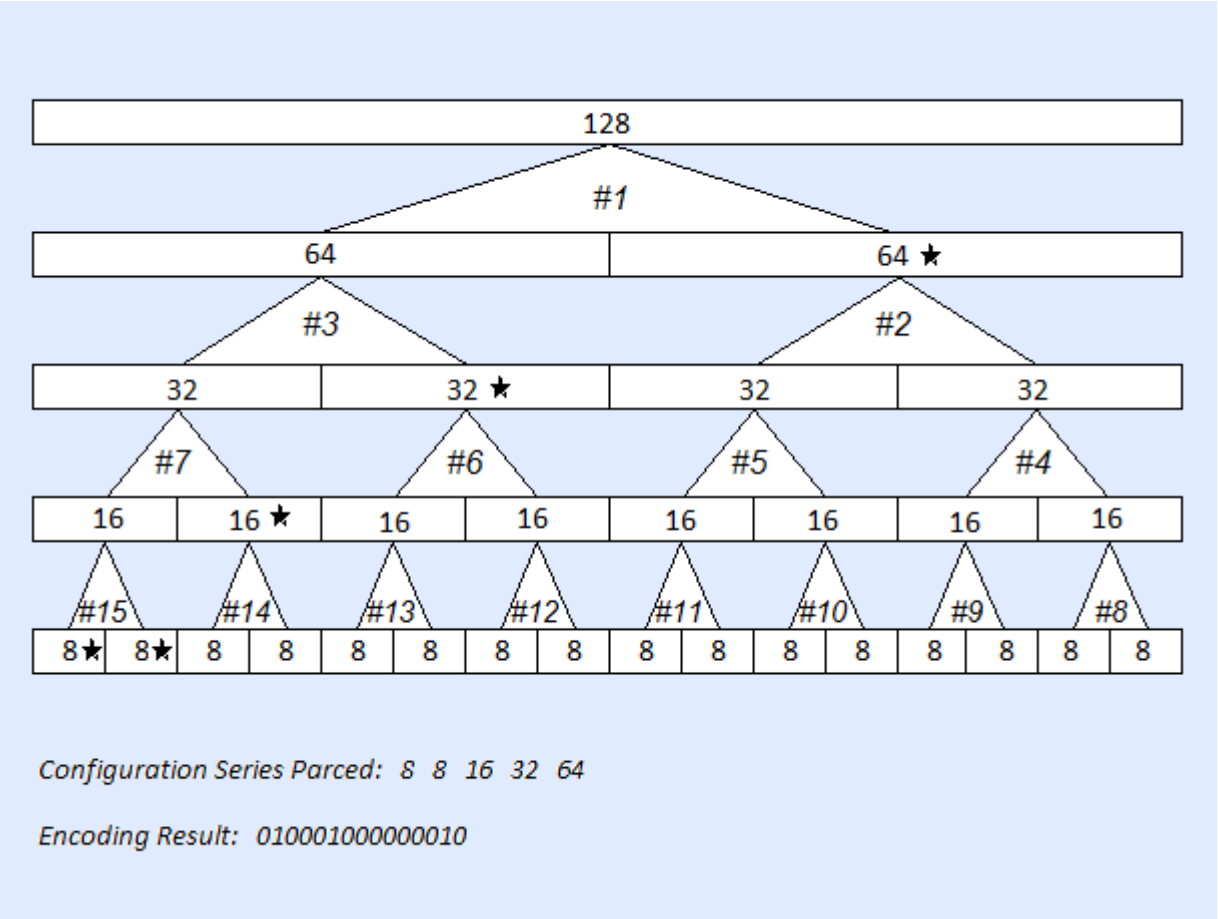

Figure 3.1: 128-bits comparator splitting, parcing, encoding

Note that a single-length comparison scheme is much simpler and its encoding requires significantly fewer bits. Considering a *N* length comparator, the comparison widths are one of the {8, 16, …, *N*}, all the same for the whole comparator. The number of the different comparison widths is $1+\log_2(N/8) = \log_2(N/4)$, or $1+1+\log_2(N/8) = \log_2(N/2)$ if we also consider the encoding of the case the comparator is not used. Thus, the encoding requires $ceil(\log_2(\log_2(N/2)))$ bits, where ceil() the rounding to the closest upper integer function. This means that for 128 bits we need only 3 bits of encoding compared to the 15 bits of the full VLC mode.

Table 3.1 illustrates a simplification of the single-length comparisons, variable-length comparisons, and variable per node/single per comparator length schemes, corresponding to 32-bits long comparators. As we can observe, the initial invalid series of comparison widths can be modified so that it can be handled by 4 comparators, with optimization criterion to perform as many comparisons as possible. We observe that the full VLC scheme achieves the highest number of comparisons due to its higher flexibility at the cost of more encoding bits/higher hardware complexity. V/SLC scheme preserves the small number of encoding bits/low hardware complexity of the SLC scheme and also achieves a number of comparisons slightly lower than the full VLC scheme. Of course this example is only a simplification of the logic of the algorithm, since the comparison widths are modified with

each iteration based on the number of bounds which are considered. However it should make clear the notion of the V/SLC scheme.

Comparator Scheme	Encoding Bits	Comparison Widths Series	Number of Comparisons
Invalid	-	32, 16, 8, 8, 8, 8, 8, 8, 8, 16, 8, …	-
SLC	2	{32}, {32}, {32}, {32},	4
Full VLC	3	{32}, {16, 8, 8}, {8, 8, 8, 8}, {16, 16},	10
V/SLC	2	{32}, {16, 16}, {8, 8, 8, 8}, {16, 16}	9

Table 3.1: 32-bits Comparators – Validation Process for 4 Subsequent Comparators

Beyond this innovative contribution and the parameters we presented already, the software exhibits additional flexibility. The purpose of most of the parameters we will discuss is to evaluate the resulting range tries with various different settings. Specifically, the algorithm has the following abilities:

- Change the Rules Set: Beyond the parametric subtractor width, which implicitly controls the frequency of application of rule 5 into the algorithm (setting subtractor width 0 results in omitting rule 5), the remaining rules can also be indicated if we desire to apply them. Rules 1 and 2 refer to the common prefix (see sections 2.2.1.1 and 2.2.1.2), with rule 2 being a generalization of rule 1. Since applying no rule for the prefixes defies the logic of the range tries we opted to create a parameter which controls which one we want to apply. Exactly the equivalent happens with rules 4 and 3, with 3 being a generalization of rule 4 (see sections 2.2.1.3 and 2.2.1.4), and thus a different parameter controls which one of the two we want to apply (applying both rules 1-2 for common prefix or similarly 3-4 for common suffix is meaningful only when the comparator width is different from the address length, which does not happen for the CMPN mode we implemented).

- Bounds Only: In case we opted for rules 1 and 4, which omit the common prefix and zero suffix respectively, there is no need to reserve part of the bandwidth for storing common prefix/common suffix respectively. A parameter controls whether we want to utilize the full bandwidth for bounds to compare or not. This produces some interesting results presented in section 3.2.4.

- Alter the Upper Bound Selection: As we mentioned in 3.1, the parameter *MINBORDERS* indicates the minimum allowed *fill factor f* of the node and affects the area of search for the most suitable Upper Bound.

- Comparator Modes Compatibility: This parameter decides between the simple comparator encoding scheme and the complex one depicted in figure 3.1. In the case of full VLC mode the complex is selected automatically since it is the only which can represent it.

- Comparator Full Utilization: This parameter is effective only for the IPv4 version. In many cases the last used comparator in a node is not fully used. This parameter activates a small

piece of code which modifies the last comparison widths in order to utilize fully the last comparator. This relieves the hardware from the work to check for zero value in the comparison field.

All the above presented parameters contribute to the versatility and flexibility of the code and enable it to handle different types of configuration files for the various hardware range tries implementations. Despite the number of different parameters the code is quite compact since the different types of configurations share a large part of the code (parametric algorithms). Performance characteristics of the code will be discussed in the following section.

3.1.3. Software Performance

Concerning the software performance, our purpose was to create software which would process successfully and fast any size of input lookup table. Since our priority is the correctness of the code and its ability to perform reliably in practice, execution speed is of secondary importance, without ignoring it either. In this section we will discuss about the software modifications performed in respect to these two characteristics.

3.1.3.1. Memory Management

Normally after ensuring the software correctness the software should be reliable. However, the size of the lookup table can be large, and for this reason if the memory management is not implemented properly (memory leak or inefficient programing with excessive memory allocations), the already large memory requirements lead the program to crash, which is unacceptable for reliable software. The difficulty of this issue for our algorithm becomes apparent from the previous work presented in [7] and [8]. Due to the inefficient memory management strategies the maximum size of the lookup table which could be handled by the software did not surpass the 4,000,000 prefixes for the IPv4 version of the code and the 1,000,000 prefixes for the IPv6 version of the code (there were different versions for IPv4 and IPv6 because 128 bit addresses cannot be handled with the standard supported formats offered by the hardware, since the longest integer is 64 bits and thus software structures have to be invoked for their representation and respective calculations). This constituted a great drawback of the previous versions and our aim was to amend it.

When we analyzed the memory allocations performed during the execution of the code we located the greatest memory requirements in two points:

The first and most significant is the memory required for the storing the nodes after they have been constructed by the software but before they have been printed properly by the output module. If we had followed the top-down approach it would be possible to print the node directly after its creation, eliminating the need for storing it. However, in our code we use the bottom-up approach, meaning that the final number of levels is unknown. Unless post-processing is performed in the resulting configuration file to invert the hardware memory-level numbering (when a node is constructed a field indicates the distance from the bottom level of nodes, while we need the distance from the root), we need to wait, store the created nodes, and eventually start printing starting from the root. Theoretically

it would be possible to maintain the counting from the bottom level and also print the nodes in reverse order (or a more complicated order as it will become apparent later), but some additional logic would be required while reading the file to pass it into the hardware. An additional drawback would be that depending on the design, the algorithm may require a second pass of the nodes before they are ready to be printed, like the splitting process described in [8]. In our code we did not implement such process but the data-flow of the code permits the integration of such a module if needed (the most effective way would be concurrently with the printing).

For all the above mentioned reasons we decided to *store the created nodes in files in the hard disk*. A first design decision was the format of the file. Considering that the node contains numbers as objects of the *mpz_class* which is suitable for representing these numbers, their actual memory size is significant (64 bytes). From those we use 128 bits, which means only 25% of their real potential. Moreover, a significant portion of those is zero. These characteristics indicate that storing the node in text format is actually more efficient than storing them as binary file (each character requires 8 bits and represents 4 bits of our number, meaning 50% efficiency, and also zero numbers are represented by a single character only). This is important because for large input lookup tables the total size of the stored data may surpass the 10 GB. An additional benefit is that we can visually inspect the intermediate results (nodes) or even alter them manually for experimentation reasons.

The second decision refers to the number of files and organization of the data in the files. As we will present shortly, the nodes are created in a complicated order but our printing function needs them ordered from the root to the leaves and from the left to the right. Considering these we opted for one file per level of nodes (automatic file naming and creation) which leads to a straightforward process of retrieving the nodes from them for printing.

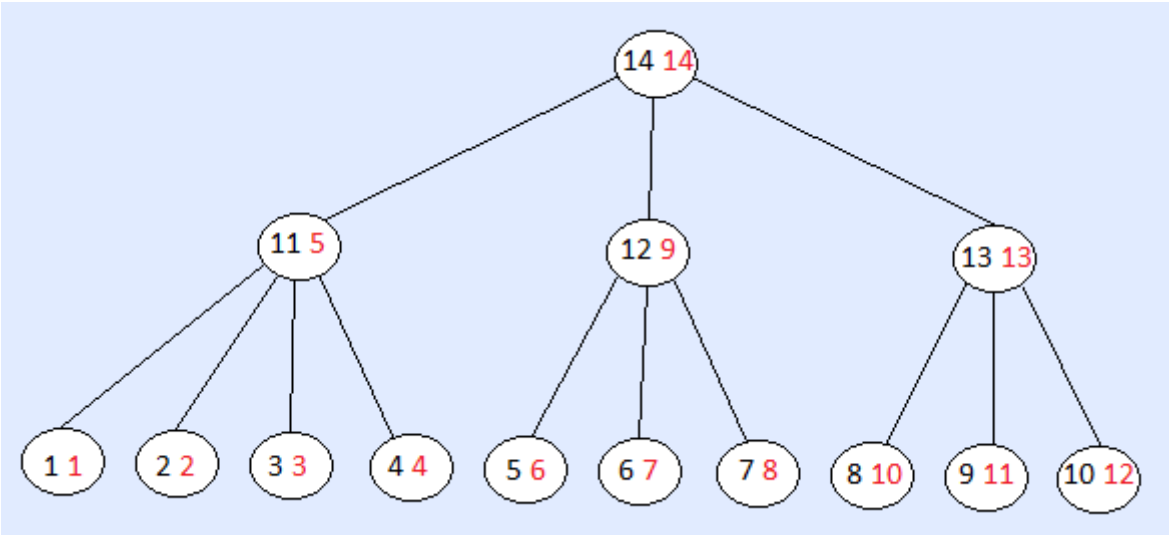

Figure 3.2: Reordered Processing Sequence

The second highest memory requirement stems from the bounds themselves. This does not refer only to the initial bounds, but also to the upper bounds which are created by the nodes themselves for comparison at their parent nodes. Rather than storing the upper bounds too, we found a better solution. Instead of processing first all the leaf nodes based on the initial bounds and creating all the

upper bounds for the next level, and this repeatedly until the root node, we eliminate completely the need for storing the bounds by *reordering the processing sequence*.

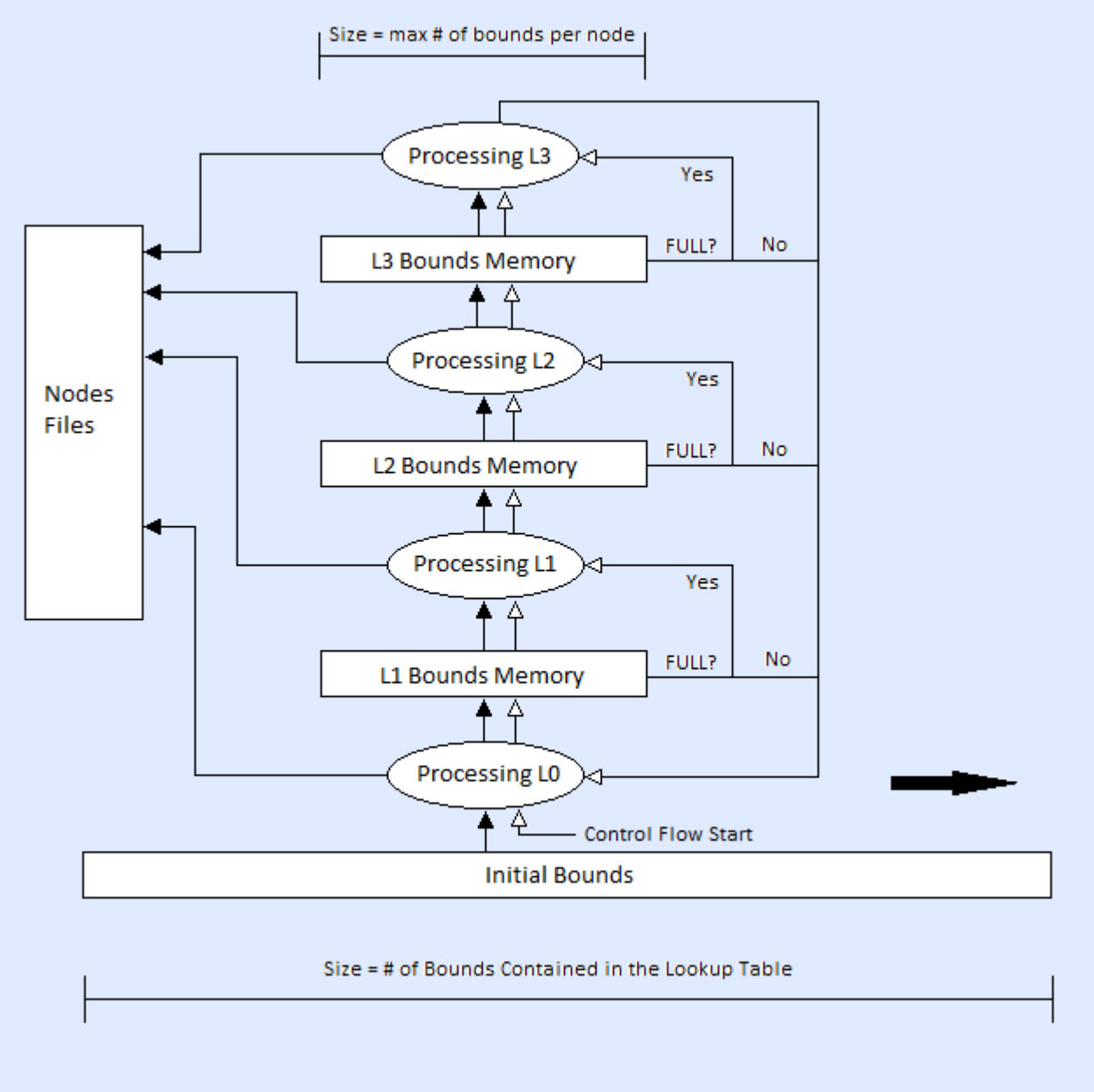

Figure 3.3: Node Processing Scheduling – Control and Data Flow

More specifically, we process the bottom level nodes until enough upper bounds have been created and then immediately process the higher level node. We keep the remaining bounds and add to them the new upper bounds created by the bottom level, and the process repeats. At the same time we perform the same for the higher levels too. An example of the order in which the bounds are processed and the respective nodes are created can be seen in figure 3.2, though it is a simplification of the real process, since in reality we cannot know beforehand the number of children of the parent node. The black numbers indicate the order in which the nodes are created in the classic bottom-up implementation, while the red correspond to our reordered bottom-up.

Thus the memory needed for the upper bounds (per level) is limited to the number of bounds a single node can process, since subsequently the same memory is reused. This minimizes extremely the memory required for the processing. In our code we preserved the initial bound array in the main memory because it has to be sorted initially and we do not consider that this comprises part of the main processing. If needed, the code can be easily modified to read the sorted initial array of bounds from file.

Figure 3.3 depicts this memory structure schematically for an example 4-levels range trie, where black arrows indicate the dataflow and white arrows the control flow. The *initial bounds* are used for creating the bottom level nodes by *Processing L0*, which also creates an upper bound and stores it in *L1 Bounds Memory*. This repeats until *L1 Bounds Memory* is full. Then *Processing L1* is activated and creates a node at the second level, passing also an upper bound at *L2 Bounds Memory*. The used bounds at *L1 Bounds Memory* are removed, and the processing can return to the initial bounds and *Processing L0*, adding upper bounds to the remaining positions of *L1 Bounds Memory* until it is again full. When that happens *Processing L1* is activated again etc. This resembles a recursive process, but the actual implementation we created is not recursive. The processing continues at all levels and for all the initial bounds until all the initial bounds have been integrated into nodes at the first level. Then, a similar *follow-up* process takes place (not shown in figure 3.3) which completes the processing by creating nodes based on the remaining bounds in the *Lx Bounds Memories* until they are all empty.

What we achieved is an algorithm which has *constant memory requirements*, independent of the number of bounds to be processed. There is still some connection between the memory bandwidth and the allocated memory, but even for 1024 bits memory bandwidth the processing memory requirements did not exceed the 1 MB. In contrast, for our experiments, the initial bounds array was the dominant factor (memory used for it is: Number_of_Bounds*64 bytes), but as we mentioned already this is not part of the processing and can be easily omitted if required.

Based on these changes we were able to run tests for both IPv4 and IPv6 with up to 25,000,000 prefixes size lookup tables, while the ability of the processing is much higher considering enough time and hard disk space.

3.1.3.2. Execution Time Reduction

The execution time for our software is also an important factor. The optimized memory management also helped from this perspective. However, the majority of the execution time is spent in the node creation heuristics. For this reason, we tried to optimize the core of the algorithm as presented in the start of section 3.1. Compared to the initial version, most of the code has been rewritten from scratch performing optimizations for speed (reduce loop bounds when possible, smarter basic algorithms, calculations repetition avoidance etc.), maintaining though the parametric nature of the algorithms. Exception is the upper bound selection process, where we did not perform any modifications. The reason for this was that the initial code was correctly and nicely implemented, thus we preserved it.

The execution time results can be seen in section 3.2, where we will perform their evaluation compared to certain parameters. Of importance as proof of our work is the comparison to previous work found in [7], considering also the fact that we perform file operations which are generally slower. It is hard though to evaluate the exact gain from algorithmic improvements considering the

differences on the systems where the code has been executed, the different input, or even possible bugs in the code which alter the timing results.

3.1.4. Range Tries Quality

In our software we tried to exploit to the highest degree the flexibility provided by the range trie structure. The software needs complicated algorithms in order to perform its functionality in an optimal way (the core of the algorithm represents a good example of it, since we try to determine the optimal configuration for a node based on the extracted information like the common prefix/suffix, subtract value, the minimum comparison widths per bound, the validation process, while all these change with any new bound added to the bound group). However, with careful programming we managed to transform the idea of the heuristics into working software which creates high quality results, and at the same time comprises a convenient testing platform for the capabilities of the range tries.

The respective results can be seen in the next section (3.2), along with various other results of the testing we performed.

3.2. Experimental Results

As we described in section 3.1.2, the software we created contains a large number of parameters which alter the resulting configuration file. These changes either refer to different hardware implementations of the range trie, or to the same hardware implementation, but the stored range trie in it can exhibit different qualities. Due to their number, extensive experimentation with all the possible parameters combinations is not feasible, especially considering the fact that the generation of the configuration file for large lookup tables may need time in the order of several hours (see sections 3.2.1 and 3.2.2).

For this reason, we examine the effect of each parameter on our algorithm and on the resulting range tries independently, usually keeping the rest stable or modifying maximum one more. This selective exploration of the problem space provides us with a deep insight into the mechanics and properties of the range tries and the applied heuristics.

The rest of this section is organized as follows. In section 3.2.1 we will present the results concerning typical possible hardware implementations of the range trie structure and we will analyze their characteristics. Subsequently, section 3.2.2 will present a comparative analysis of the variable-length comparisons scheme with the single-length comparisons scheme and the variable per node/single per comparator scheme which we presented in section 3.1.2. In section 3.2.3 we will present the results we obtained when we measured the effect of each rule on the heuristic. Based on these results we attempted to examine the potential of a new version of the range trie, named *simplified range trie*, and the respective results are presented in section 3.2.4.

3.2.1. Highest Quality Range Tries

Our first priority for experimentation is to measure the quality of the results obtained by the current range trie structure and the developed heuristics which create the initial configuration. Since we want to obtain the highest quality results for some (practical or theoretical) range trie hardware implementations, we provide to the heuristic parameters matching those of the hardware and at the same time we define the remaining parameters with values which allow the heuristic to perform the maximum number of optimizations (in respect to quantity and quantity).

We consider three different typical hardware implementations. The first refers to IPv4 addresses and its memory bandwidth is 256 bits (red color and dotted line in the following figures unless otherwise mentioned). This is actually an existing implementation, whose last and most complete version is presented in [5]. The second refers to a theoretical but feasible implementation for IPv6 addresses whose memory bandwidth is also 256 bits (blue color and dashed line in figures), while the third also refers to IPv6 addresses but with 1024 bits memory bandwidth (black color and solid line in figures). Note that the third implementation preserves the same ratio address length/memory bandwidth to the first implementation which makes them interesting for comparison. However, creating memory bandwidth of 1024 bits is extremely challenging up to infeasible for the current hardware technology. We consider that all of the three implementations contain 8-bit long adders for the address alignment, while also all the comparators of the designs have the ability to perform up to 8-bits long independent comparisons as presented in section 3.1.2 (see figure 3.1).

Considering these characteristics, we set the proper heuristic parameters for the address length and memory bandwidth, we activated rule 5 (address alignment) with 8 bits long maximum subtract value width, we declared that we desire the VLC version, and we utilized all the remaining rules (which means 2 and 3 as generalization of 1 and 4 respectively, and further optimization with 4 during the VLC computations phase). Based on these parameters, we created the respective range trie configuration files for various sizes of the lookup table (number of prefixes/bounds) and we obtained the following results.

In figure 3.4 we can observe the number of range trie levels for a great range of lookup table sizes for these three implementations. From this we can already confirm the great potential of the range trie structure when compared to other relevant address lookup/longest prefix match techniques. For instance, considering the generated range trie configuration for the IPv6/256 bits memory bandwidth implementation, we observe that it requires only 8 range trie levels for 16 million prefixes (approximately 29,500,000 bounds based on our random prefix generator). Comparatively, a balanced range tree for IPv6 and 256 bits memory bandwidth would require a minimum of 16 levels for the same number of prefixes, and if the processing time per level for the two schemes, the range trie would have 50% decreased latency. On the other hand, a k-stride multi-bit trie would require k=16 to achieve the same number of levels as our range trie, which is a very large number for that scheme.

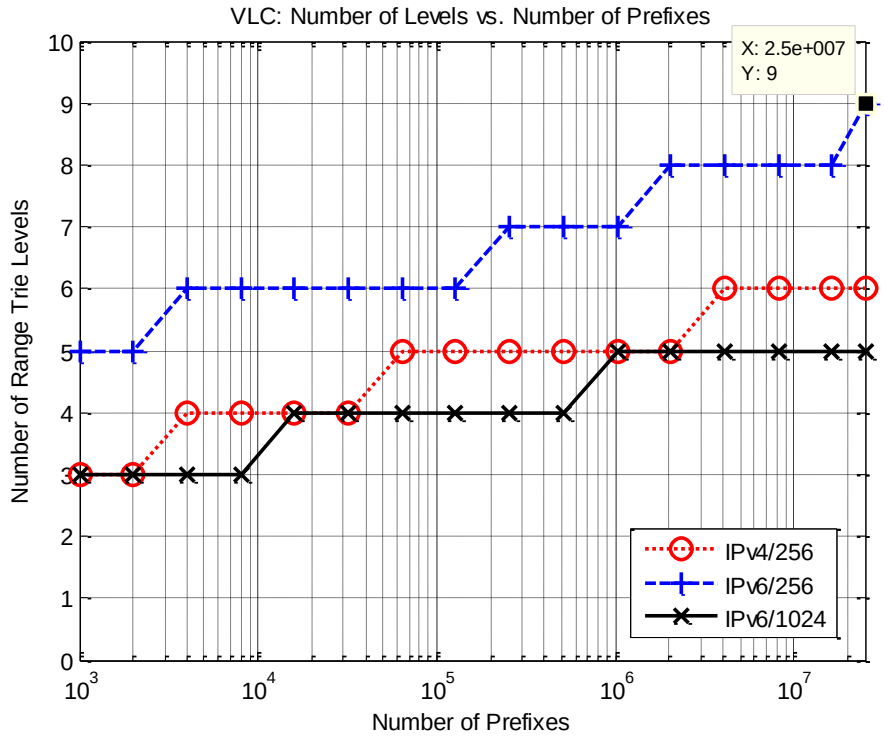

Figure 3.4: VLC: Number of Levels vs. Number of Prefixes

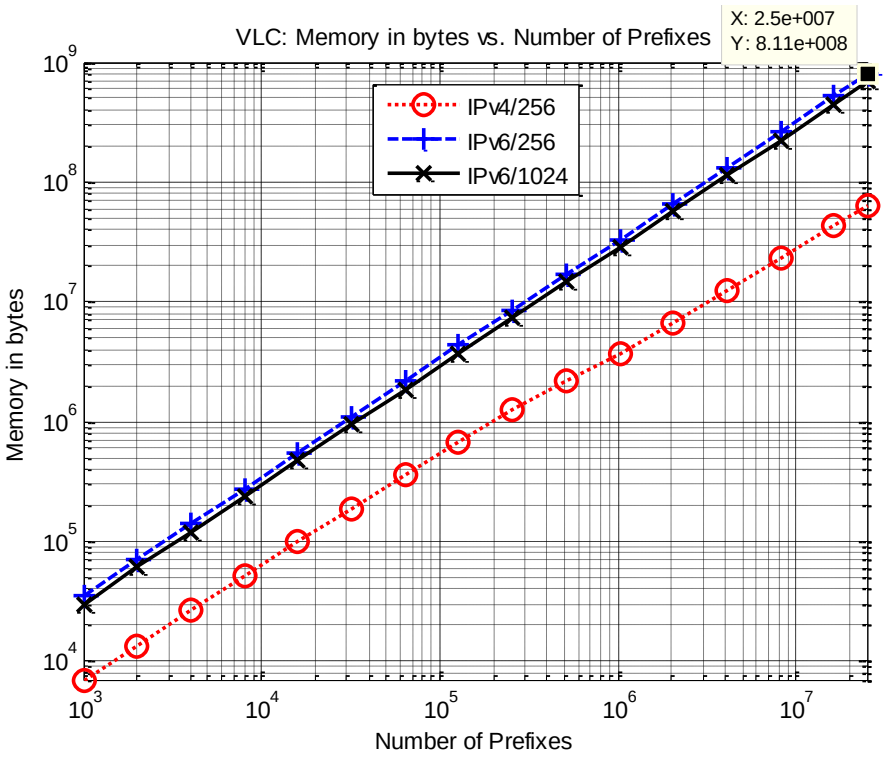

Figure 3.5: VLC: Memory in bytes vs. Number of Prefixes

Another important attribute to examine is the memory requirements. Figure 3.5 presents the memory requirements for the three distinct hardware implementations and various sizes of the lookup table. For fairness reasons, since the three hardware implementations do not have the same memory bandwidth, it is preferable to measure the required memory in bytes rather than number of nodes. Based on these results we can make two major observations: First, for the IPv6 versions, the required memory rises approximately proportionally to the number of prefixes, though for high number of prefixes we can observe a slight deviation from the linearity (part of the reason behind this is that the number of bounds does not increase exactly proportionally to the number of prefixes, but we reach the same conclusion even considering the memory requirements compared to the number of bounds). This phenomenon becomes even more apparent for the IPv4 version. The reason behind this is the branching factor of the first level nodes as it will be presented shortly. Second observation when comparing the two IPv6 versions is that the one with the higher memory bandwidth has consistently lower memory requirements. There are two reasons for this; first, the memory required by the higher level nodes (after the first) is significantly less for the 1024-bits bandwidth version (see tables 3.3 and 3.4), and second, with higher bandwidth the bandwidth utilization increases, as we will explain in section 3.2.4.

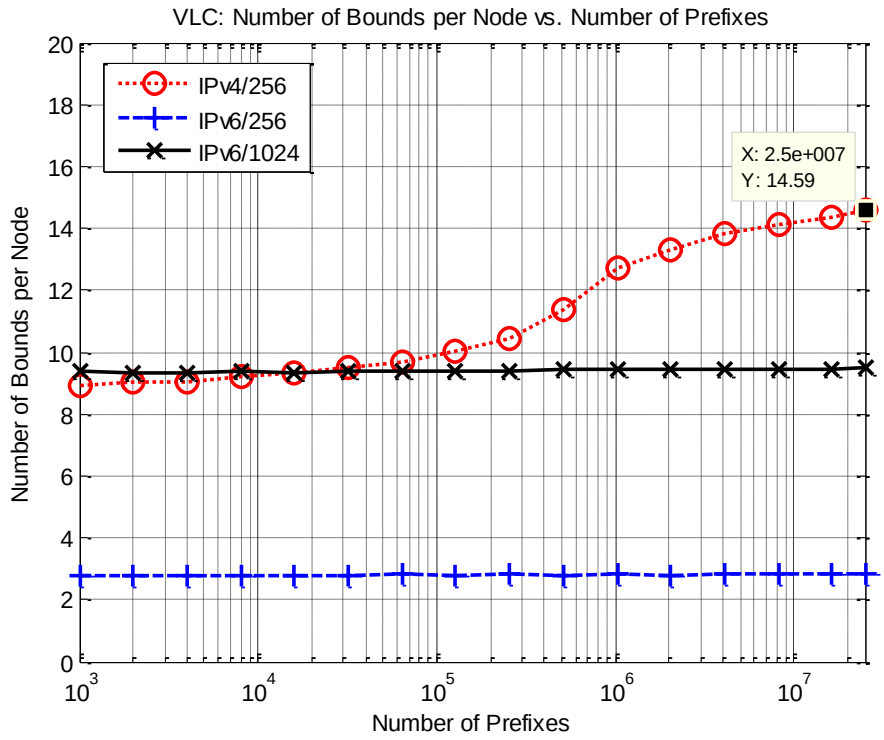

Figure 3.6: VLC: Number of Bounds vs. Number of Prefixes

Figure 3.6 shows the average number of bounds per node for the three hardware implementations and various lookup table sizes. We can observe, intensely for IPv4 and very slightly for IPv6, that with higher number of prefixes, the range trie structure increases the number of bounds per node and starts revealing its true potential.

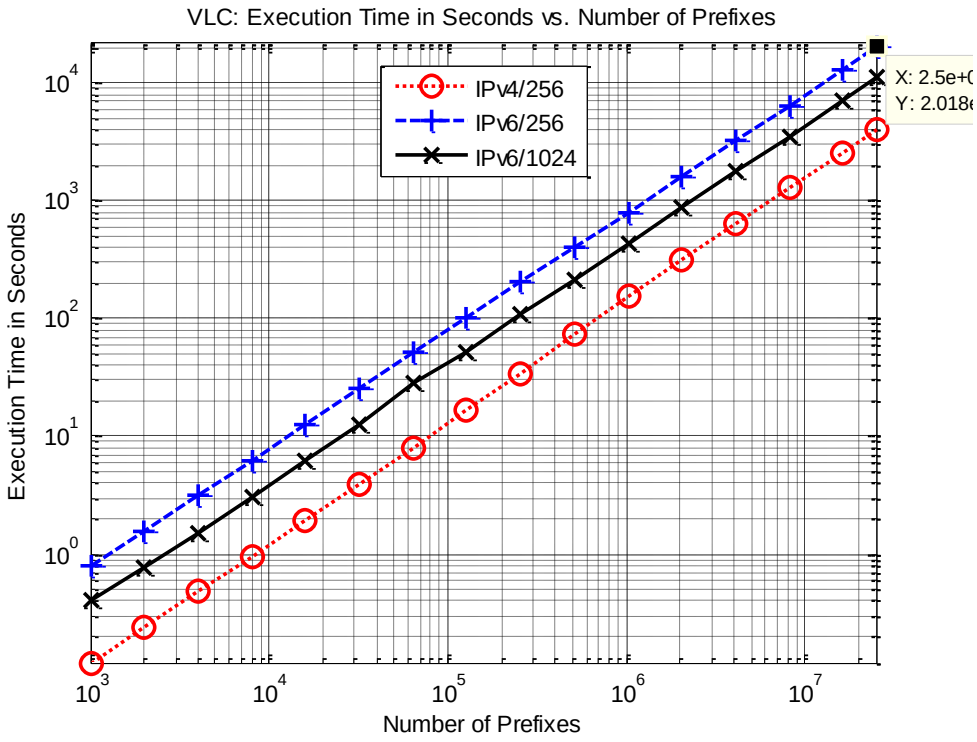

Figure 3.7: VLC: Execution Time vs. Number of Prefixes

The time required for the generation of the respective configurations is presented in figure 3.7. As we can observe, many parameters affect it significantly; the address length, the size of the lookup table (note the minimal deviation from linearity), the memory bandwidth (which affects the node size/number), and based on the results from section 3.2.2, the version of the algorithm.

Figures 3.8 through 3.10 contain the number of nodes of the range tries generated for the three implementations for lookup table sizes: 1, 2, 4, 8, 16, 25 million prefixes (black, red, green, cyan, blue and magenta respectively starting from the bottom part of the figures). The respective data can be found in appendix B. From these we can extract several of the characteristics we presented earlier.

Related to these tables and the previous figures are the figures 3.11 through 3.13 where we can see the average branching factor per level for the three implementations. Specifically, we can observe that for the bottom levels the branching factor is relatively lower since the initial bounds are “inconvenient”. The middle levels have much more “convenient” bounds to process since they were produced by the algorithm itself with this purpose in mind, and thus the branching factor is higher. For the top levels (root or close to root), the results for the branching factor are inconsistent due to the small number of nodes at that levels.

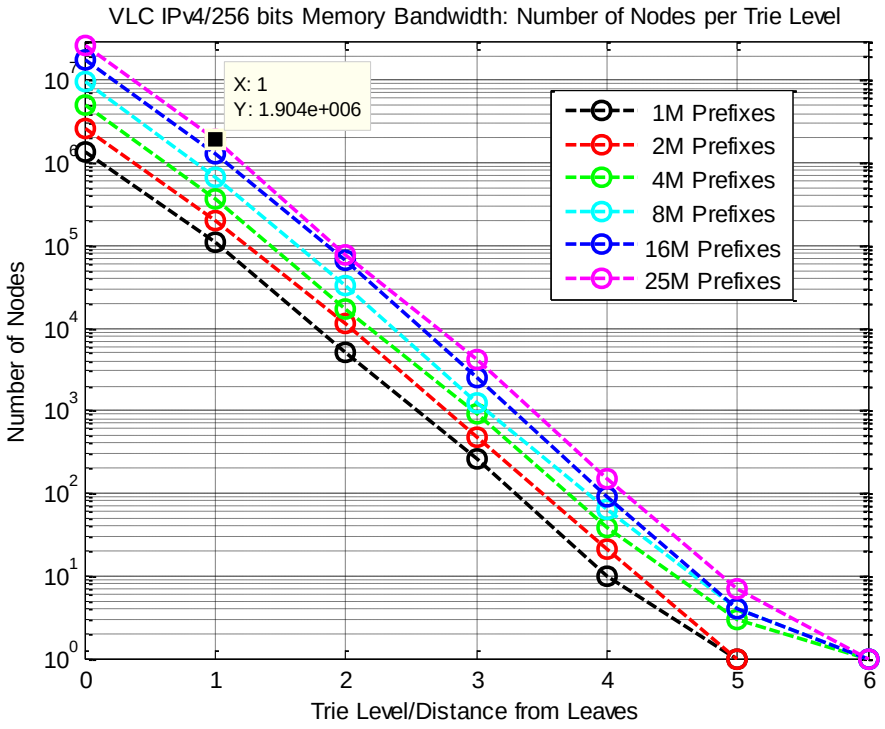

Figure 3.8: VLC IPv4/256 bits MEMBW: Nodes per Level

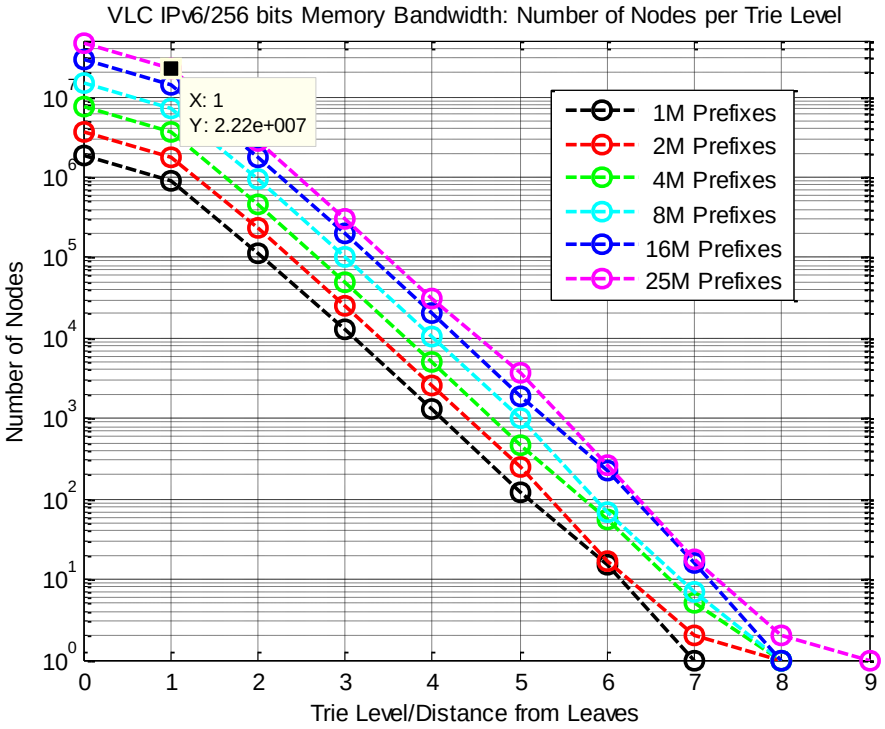

Figure 3.9: VLC IPv6/256 bits MEMBW: Nodes per Level

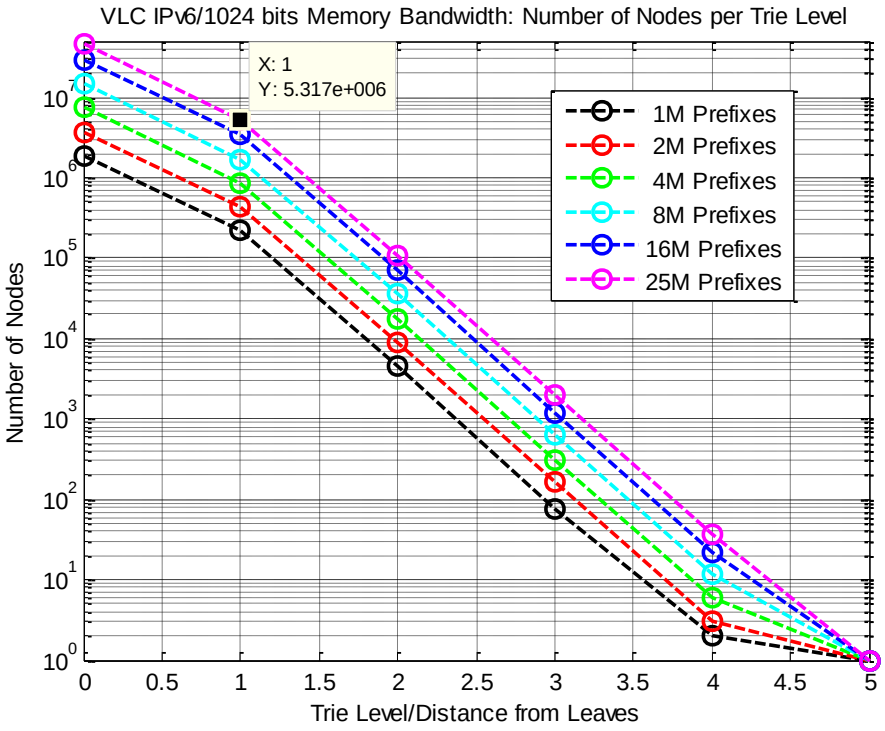

Figure 3.10: VLC IPv6/1024 bits MEMBW: Nodes per Level

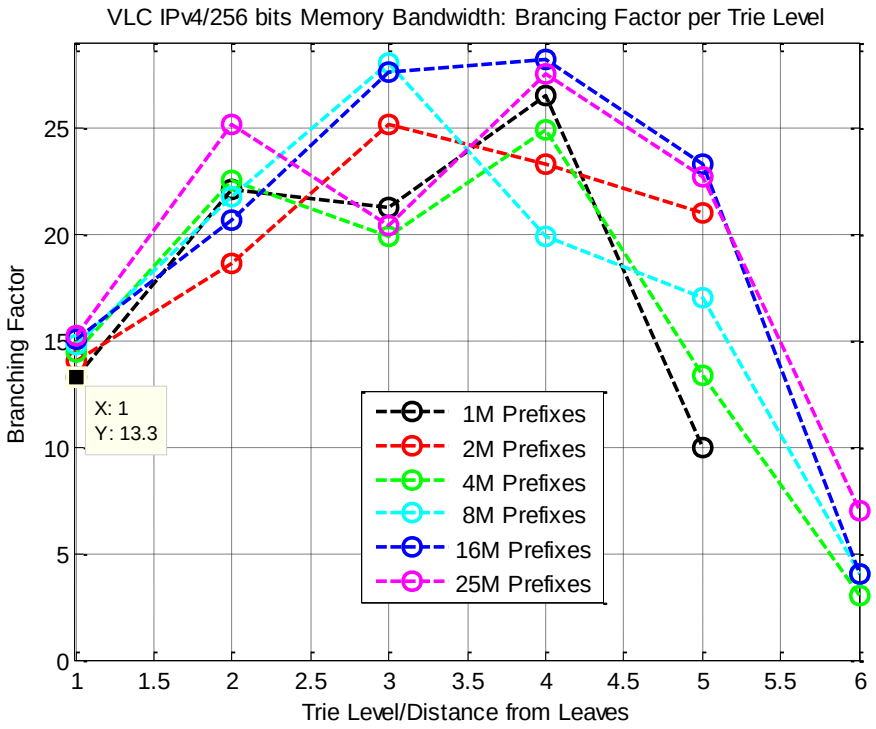

Figure 3.11: VLC IPv4/256 bits MEMBW: Average Branching Factor per Level

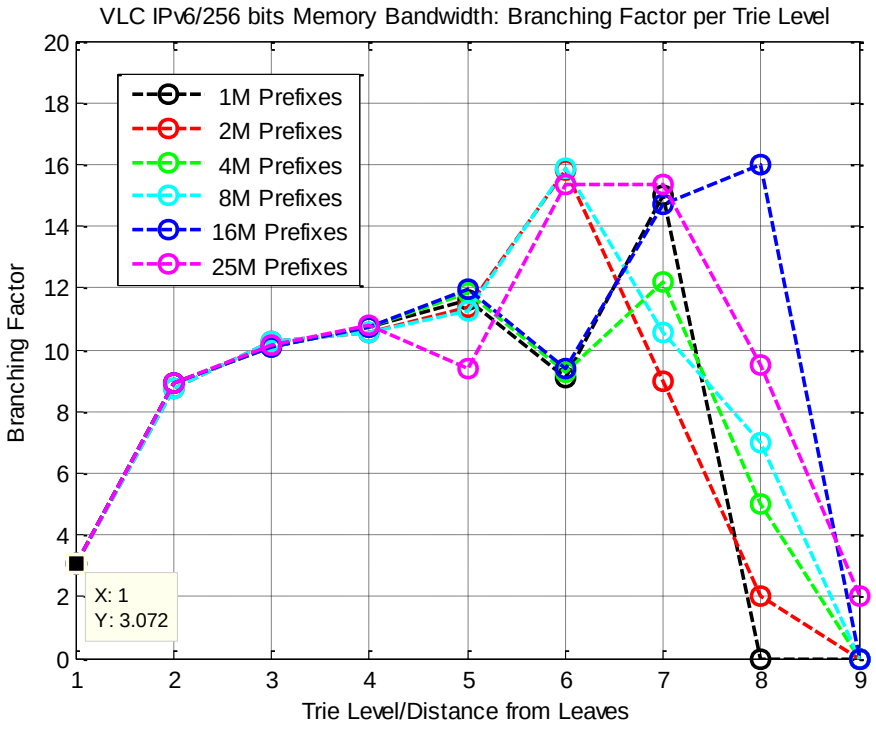

Figure 3.12: VLC IPv6/256 bits MEMBW: Average Branching Factor per Level

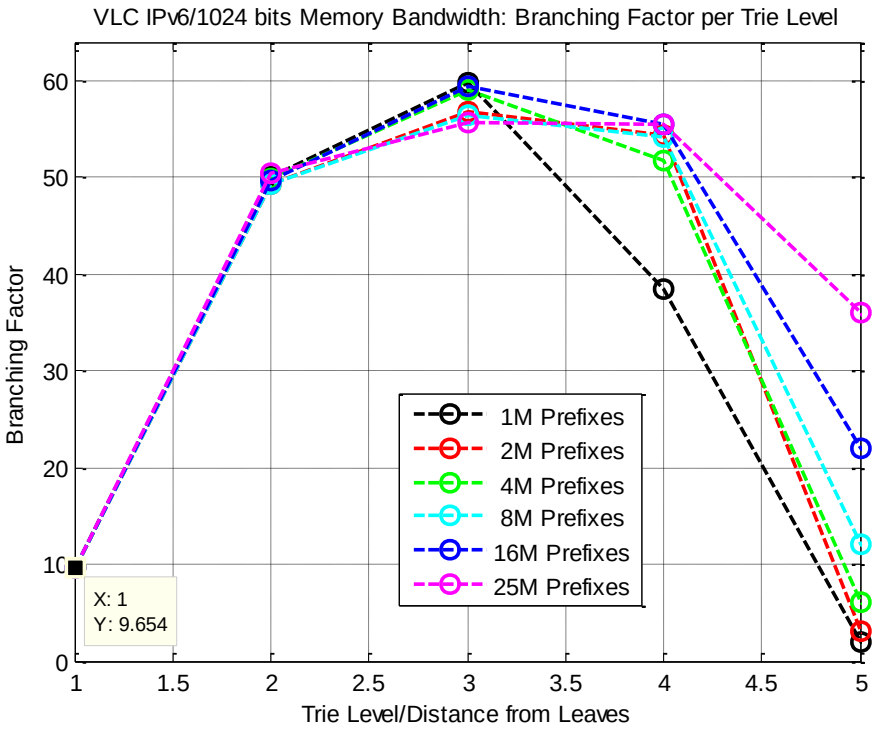

Figure 3.13: VLC IPv6/1024 bits MEMBW: Average Branching Factor per Level

This concludes with the results concerning the quality of the range trie structure. According to those we can already observe that the range trie is indeed an address lookup scheme with great potential, and the created heuristics are fully capable to support its complexity. In the following sections we will further present experimental results and analyze the characteristics of the range tries.

3.2.2. Comparison of VLC – V/SLC – SLC Range Tries

In section 3.1.2 we presented the meaning of variable-length comparisons per node and single-length comparisons per node range tries. Their effect on the results was also discussed in [7] and based on the results presented there, the VLC scheme had a great advantage over the SLC scheme and thus the higher complexity of a VLC range trie hardware implementation was deemed to be worth it.

In this section we will present our own experiments concerning this feature. In addition to the work in [7] and according to the description we offered in section 3.1.2, we also experimented with the *variable length per node/single per comparator* comparisons scheme (V/SLC). We chose for experimentation the two technologically implementable hardware versions; IPv4 and IPv6 with 256 bits memory bandwidth. Note however that for IPv6 with 256 bits memory bandwidth, each node has two 128-bits long comparators, from which the second can only be used to perform a full 128-bits comparisons and this only if the first comparator is also used for a full 128-bits comparison. For this reason the respective results for SLC and V/SLC totally coincide since we have only a single comparator able to perform variable comparisons.

Figure 3.14 shows the number of levels of the generated range tries for various sizes of the lookup tables concerning the IPv4/256 bits memory bandwidth implementation. The red dotted line now represents the VLC version, the blue dashed line represents the SLC, while the black solid line represents the V/SLC. As we can observe, even though the encoding requirements for the V/SLC are as low as for the SLC (see section 3.1.2), the results it obtains level-wise totally coincide with the superior results of the VLC (based on the data points we selected as they can be seen in figure 3.14). The respective results for the IPv6 version can be seen in figure 3.15, where note that blue (SLC) and black (V/SLC) coincide as we predicted earlier.

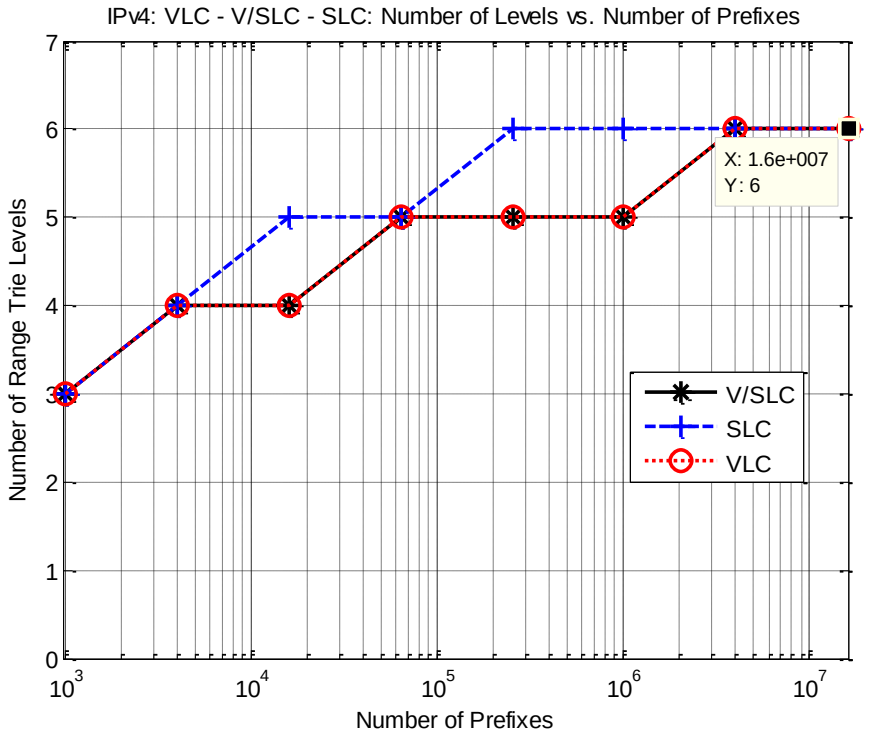

Figure 3.14: IPv4: VLC – V/SLC – SLC: Number of Levels vs. Number of Prefixes

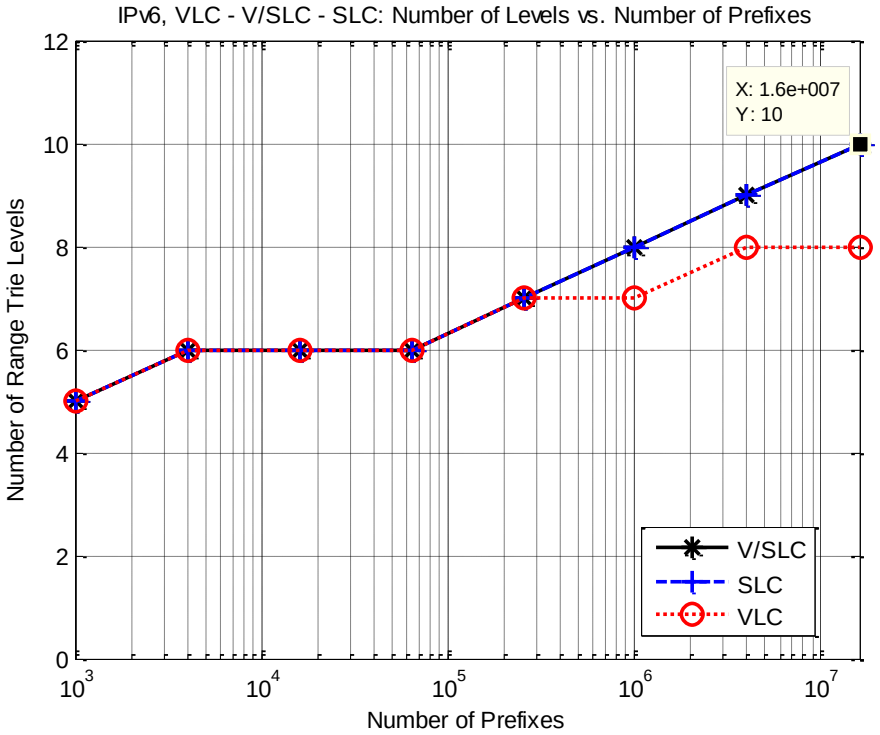

Figure 3.15: IPv6: VLC – V/SLC – SLC: Number of Levels vs. Number of Prefixes

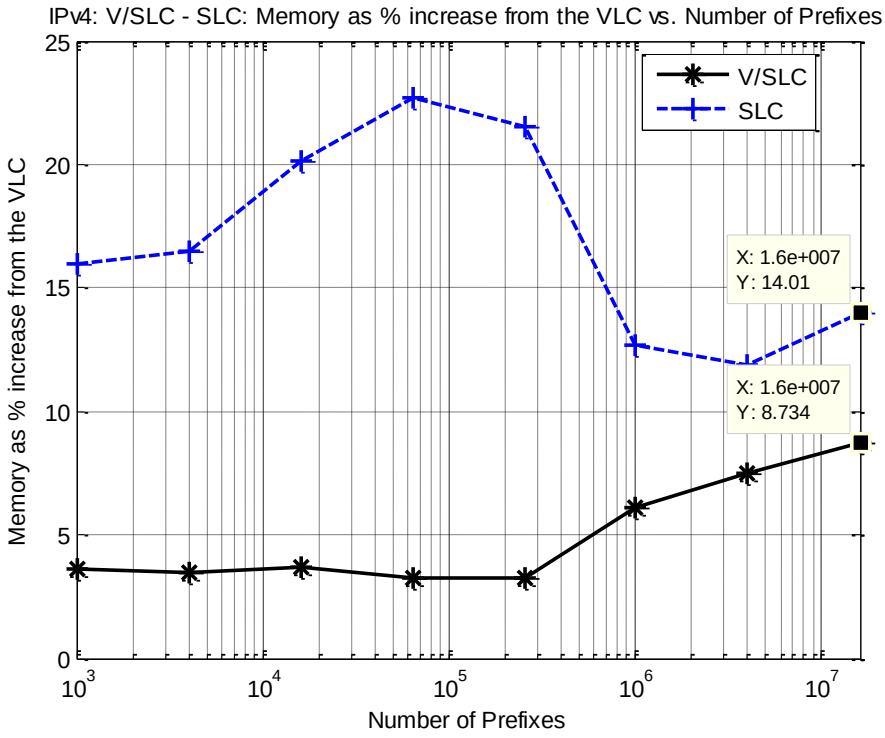

Figure 3.16: IPv4: V/SLC – SLC: Percentage Memory Increase vs. Number of Prefixes

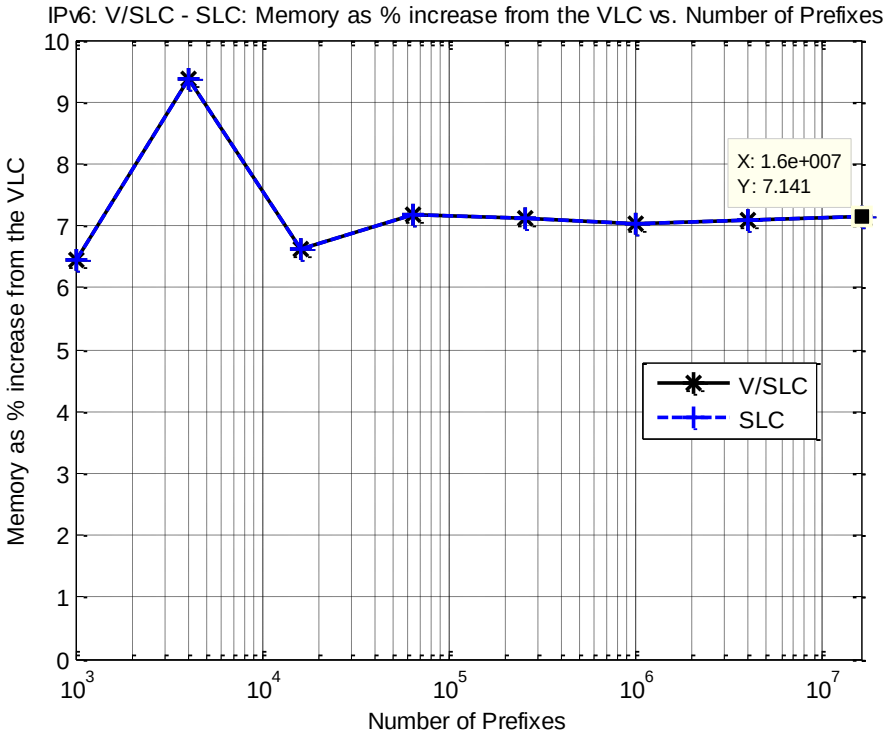

Figure 3.17: IPv6: V/SLC – SLC: Percentage Memory Increase vs. Number of Prefixes

Figures 3.16 and 3.17 indicate the memory requirements of the SLC and V/SLC schemes for the IPv4 and IPv6 versions respectively. The actual memory sizes for the VLC version can be seen in figure 3.5. Rather than expressing the memory requirements in actual values, which would make it hard to discern the differences, we express the memory requirements for the SLC and V/SLC schemes as *percentage increase* from the baseline memory requirements of the VLC scheme. From figure 3.16 we can observe that the SLC scheme requires increased memory between 12% and 23% compared to the VLC scheme, while the V/SLC scheme requires consistently lower memory in comparison (increase only between 3% and 9% compared to the baseline VLC). The shape of the curves has to do with the mechanics of the heuristics, since there are thresholds which enable certain optimizations. However, this problem is a heavily stochastic process since the complexity of the algorithm combined with the characteristics of the random input produce results which are extremely hard to trace back to their cause. For example, by increasing the number of prefixes at some point, a certain bounds-density threshold may be surpassed and an optimization which was not available for fewer prefixes becomes now available. In contrast to the IPv4 version, the IPv6 version appears to have mostly stable increase in the memory requirements for the SLC and V/SLC schemes at approximately 7%.

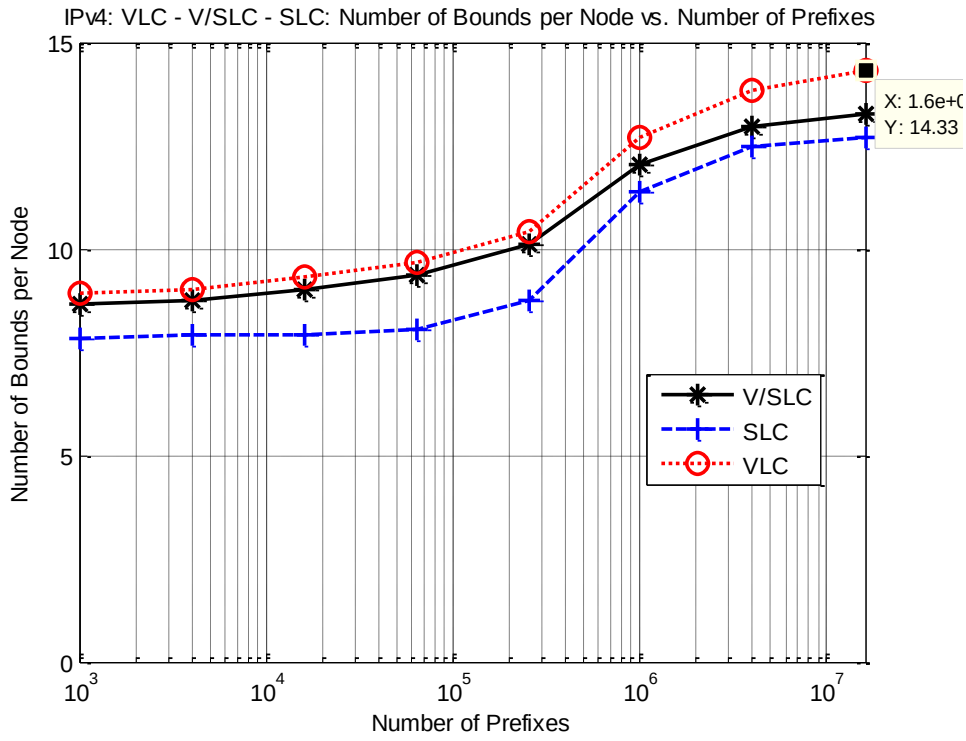

Figure 3.18: IPv4: VLC – V/SLC – SLC: Number of Bounds per Node vs. Number of Prefixes

Figure 3.18 shows the average number of bounds per node for the VLC (red), SLC (blue) and V/SLC (black) schemes for the IPv4 version. Again we observe the superior results of V/SLC compared to the SLC. Figure 3.19 shows the same information but for the IPv6 version. Note that the average number of bounds per node remains the same for the latter. This happens because the vast majority of the nodes belong to the first level and in most cases the heuristic does not manage to integrate more

than two bounds per constructed first level node since the number of bounds compared to the address length is still small and thus no long common prefixes appear (low bound density in the address space, the threshold which will allow shorter comparisons has not been surpassed with the tested number of prefixes). On the other hand, this does not apply to the IPv4 version, since in the region around 1 million prefixes we observe a steep increase in the number of bounds per node.

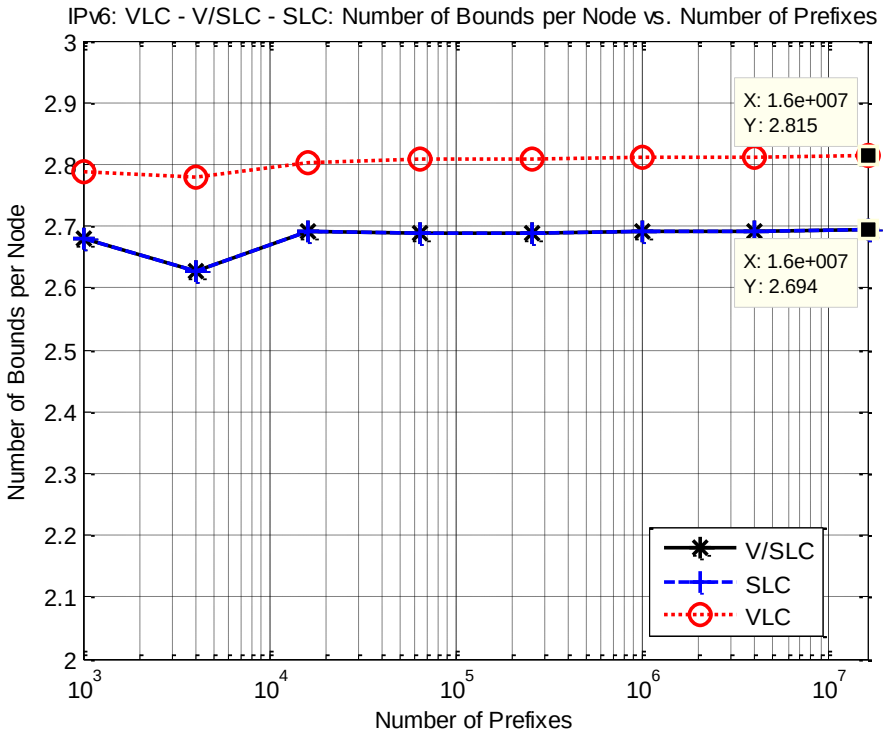

Figure 3.19: IPv6: VLC – V/SLC – SLC: Number of Bounds per Node vs. Number of Prefixes

Lastly, the respective execution time measurements can be observed in figures 3.20 and 3.21 for IPv4 and IPv6 respectively. Generally there are two conflicting elements concerning this; the increased complexity of the VLC and V/SLC tends to make each node generation slower, but also results in a smaller total number of nodes. The latter partly offsets the first element, but overall there is a tension for the VLC and V/SLC to be approximately 15-20% slower than the SLC, while VLC and V/SLC require approximately the same.

This concludes the experimental results concerning the VLC, SLC and V/SLC schemes. In the next section we are going to present testing results concerning the effectiveness of the rules integrated into the heuristics and the range trie structure.

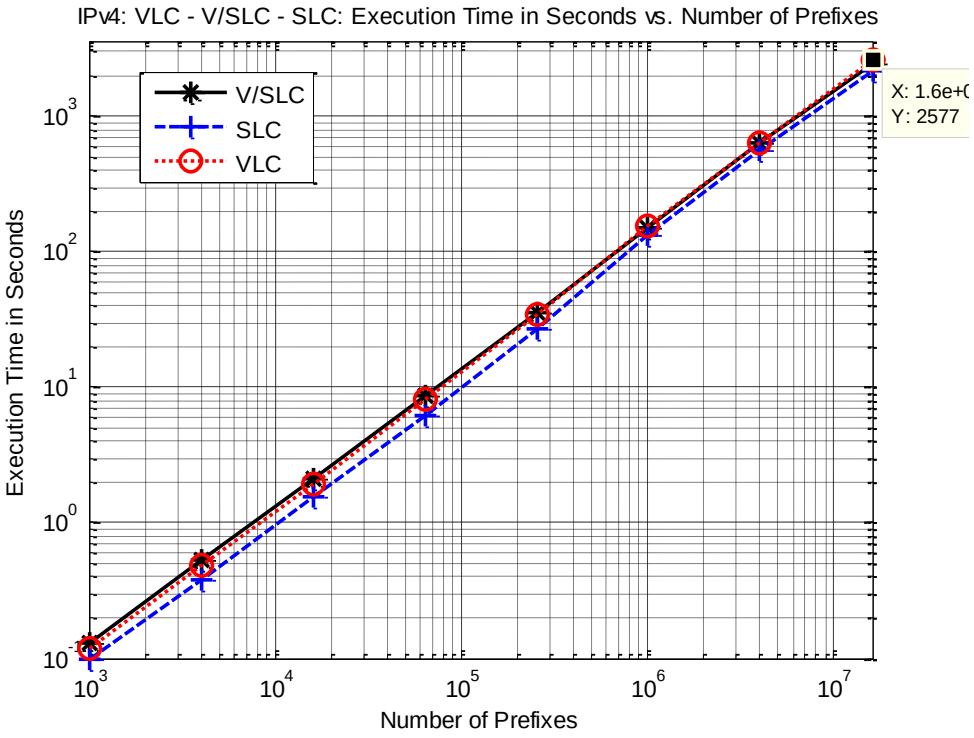

Figure 3.20: IPv4: VLC – V/SLC – SLC: Execution Time vs. Number of Prefixes

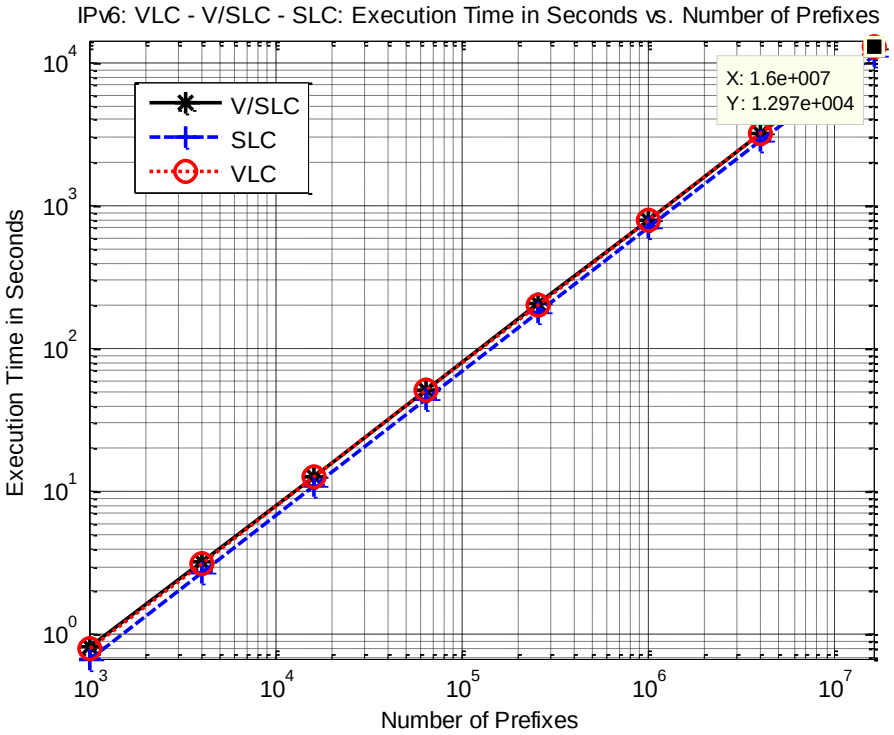

Figure 3.21: IPv6: VLC – V/SLC – SLC: Execution Time vs. Number of Prefixes

3.2.3. Rules' Effectiveness Measurements

In section 2.2.1 and its subsections we described the five range trie optimization rules. As we mentioned there, rule 2 (share comparison bounds' common prefix) is a generalization of rule 1 (omit node's bounds common prefix), and equivalently for rule 3 (share comparison bounds' common suffix) and rule 4 (omit comparison bounds' zero suffix). Rule 5 (address alignment) is applied independently after the rules concerning the suffix and before the rules concerning the prefix. In this section we will measure the effectiveness of these five rules in various combinations.

Examining the prefix and the suffix represents a core element of the range trie; without common prefixes or suffixes the range trie structure degrades into a range tree structure. Rule 5 is supplementary, so it is up to us to apply it in hope of obtaining longer common prefixes or not. On the other hand, we have to use either 1 or 2, and either 4 or 3. Considering the logic or the range trie rules and the heuristics constructed based on them, the highest quality results (least number of levels and nodes) should be obtained when the more general rules 2 and 3 are applied along rule 5. This is considered for our experimentation the base line, and all the comparative results refer to it unless stated otherwise. However, rule 2 (3) requires part of the bandwidth which is shared with rule 3 (2), and thus if we use one of the two (2 or 3), we might as well use both since they do not have any additional requirements.

Again we used for testing IPv4 and IPv6 versions with 256 bits memory bandwidth, VLC scheme. Also, we limited our testing to 1 million prefixes which is a large enough size for lookup table to be representative, but does not require extreme amounts of time to test.

Tables 3.2 and 3.3 contain the results which we extracted internally from the heuristics for the IPv4 and IPv6 version respectively. Specifically, the first column indicates which rules were applied for the results of the respective row, and the second column contains the number of constructed range trie nodes for which the application of rule 1 would result in a different common prefix length compared to the application of rule 2. Exactly equivalently, the third column contains the number of constructed nodes for which the application of rule 4 would result in a different common suffix length compared to the application of rule 3, while the fourth column contains the number of nodes for which the application of rule 5, considering also the constraint concerning the adder width (8 bits available for the subtraction) would result in a non-zero subtract value, which means that it can potentially have an effect on the later calculated common prefix length. Lastly, the fifth column contains the total number of nodes for the constructed range trie. For second through fourth columns the respective number of nodes is also expressed as percentage of the total number of nodes with the value contained in the parentheses.

For the IPv4 version we observe that rules 2 and 3, which refer to sharing the common prefix/suffix, very rarely (~ 0.01%) would have a different calculated length compared to their counterpart rules 1 and 4, which refer to omitting the common prefix/suffix. On the other hand, rule 5 seems to produce a non-zero subtraction value much more frequently, and thus we would expect it to have larger effect on the resulting range trie, even though the frequency of appearance is also fairly low (less than 2%). Note however that this is a quantitative analysis only; the qualitative will be presented shortly. Compared to the IPv4 version, the IPv6 version displays extremely different results. Rules 1 and 2 seem to produce different common prefix lengths for a very high percentage of the constructed nodes

(~ 72-73%), while the respective frequency for rules 3 and 4 is higher than the IPv4 version but still very low (~ 0.08%). Moreover, the respective percentage concerning rule 5 rises too (~ 6.8%).

Applied Rules	Rule 1 ≠ Rule 2	Rule 3 ≠ Rule 4	Rule 5 Active	Total Number of Nodes
#2, #3, #5	12 (0.0105%)	11 (0.0096%)	2289 (1.9956%)	114702
#2, #3	12 (0.0104%)	10 (0.0087%)	2292 (1.9901%)	115171
#1, #4, #5	12 (0.0105%)	11 (0.0096%)	2289 (1.9956%)	114702
#1, #4	12 (0.0104%)	10 (0.0087%)	2292 (1.9901%)	115171

Table 3.2: IPv4: Number of Nodes with specific rules Properties

Applied Rules	Rule 1 ≠ Rule 2	Rule 3 ≠ Rule 4	Rule 5 Active	Total Number of Nodes
#2, #3, #5	757944 (72.875%)	881 (0.0847%)	70627 (6.7906%)	1040063
#2, #3	757883 (72.446%)	865 (0.0827%)	70652 (6.7536%)	1046140
#1, #4, #5	757814 (72.844%)	882 (0.0848%)	70666 (6.7927%)	1040325
#1, #4	757752 (72.414%)	866 (0.0828%)	70691 (6.7555%)	1046423

Table 3.3: IPv6: Number of Nodes with specific rules Properties

The reason behind these differences becomes apparent if we examine the characteristics of the nodes as presented in section 3.2.1. According to figure 3.6, the average number of bounds per node for 1 million IPv4 prefixes is approximately 13, while the respective number for IPv6 is 2.8, with the vast majority of the bottom level nodes to contain only two bounds and the higher level nodes generally to contain much higher number of bounds. These two bounds are mostly complementary bounds created from the same prefix and thus they share all but the few last bits before the remaining zero-bits (referring to the positions beyond the prefix length). Rule 2 then considers only these two bounds and calculates a very high common prefix length. Compared to that, rule 1 also considers the node bounds (and specifically the lower because the higher has not been calculated yet for that node since the *upper bound selection* process is applied later), which are generally constructed trying to increase as much as possible the zero-suffix length, and thus the node bounds do not necessarily share as long common prefix with the two complementary bounds as these bounds share themselves. The nodes constructed for the IPv4 versions only in special cases contain less than 8 bounds, and thus statistically the common prefix calculated does not decrease if we also consider the node bounds too since it is based on independent bounds anyway (bounds not extracted from the same prefix).

The issue of infrequent differences between rules 3 and 4 is more elementary. The basic reason is that *the initial bounds never share longer common suffix than their common zero suffix.* This becomes apparent if we think about the fact that the bounds are constructed based on prefixes and this gives them special properties:

- The common suffix of two bounds extracted from the same prefix have by default *common suffix* equal to their *common zero suffix* equal to *address length minus prefix length*.
- Shorter prefixes which cover other longer prefixes define bounds for which, a bound of the shorter prefix and a (different) bound of the longer prefix have *common suffix* equal to their *common zero suffix* and equal to *the zero suffix of the bound belonging to the longer prefix* since it belongs in the interval defined by the shorter prefix.
- *A prefix either completely covers another prefix or they have no common interval*. This means that if we consider *more than two consecutive bounds*, then *definitely at least two of them belong to the same prefix or at different but covering prefixes* (a prefix corresponding to one of the bounds covers a prefix corresponding to another of the bounds).

The above three points have as a consequence that the initial bounds always have common suffix equal to their common zero suffix. A very rare exception to this rule would appear if we integrated rule 5 into the algorithm and it would create an artificial common suffix where before we had common zero suffix, but considering that the subtraction value is based on the low node bound, and this is selected based on the *upper bound selection* process of the previous node with purpose to have as long zero suffix as possible, it would still not affect the above statement. In any case, in our algorithm we apply rules 3 (or 4) before rule 5 since it is more efficient and the result is the same, and thus we eliminate this case.

Thus, the only case that common prefix is longer than the common zero prefix is during the construction of the higher level nodes (after the first), for which the processed bounds do not correspond to prefixes but are constructed by the *upper bound selection* process of the previous level nodes. Still, since the upper bound selection process tries to maximize the zero suffix, its application is rather rare. Statistically though, the fewer the bounds in the node the higher the probability for common non-zero suffix, and thus is explained the higher frequency of observing different common suffixes and common zero suffixes for the IPv6 version compared to the IPv4 (third column of tables 3.2 and 3.3).

On the other hand, rule 5 appears more often, especially in the IPv6 version, since the low number of bounds per node means high probability for longer common (zero) suffixes and thus, considering also the small range of the node bounds, it is more frequently applicable.

The above quantitative analysis though does not convey any information about the practical effect of the rules on the results. Tables 3.4 and 3.5 contain more qualitative information, since from those we can observe the number of nodes per level of the constructed range tries and thus implicitly the number of levels too. Concerning the IPv4 version, see table 3.4, rules 2 and 3 seem to bear no positive effect compared to their counterpart rules 1 and 4. On the other hand, rule 5 appears to have some minor effect on the total number of nodes per level after the first, with emphasis on the middle levels. In this case there is no difference on the number of levels, but theoretically it could lead to lower number of levels. However, concerning the IPv6 version, rules 2 and 3 have more profane effect over rules 1 and 4, by decreasing the total number of nodes and also managing to decrease the total number of levels. Note that even though at level 6 there were in both cases 15/16 nodes, meaning

14/15 bounds, rule 3 was superior to rule 4 and managed to integrate all these bounds into one node (this was observed from more detailed results not presented here, we will simply refer that rule 2 had no effect over rule 1 because the bounds themselves had no common prefix since they spanned in the whole address space). Even though rules 1 and 2 result in different common prefix lengths, this does not mean that it is possible to integrate more bounds into the same node as it is apparent from the results. As we mentioned earlier, two bounds from the same prefix have long common prefix length, but attempting to add an additional bound into the node results in short common prefix for both rules 1 and 2 and thus failure of the attempt.

Applied Rules	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5
#2, #3, #5	1343998	109255	5180	256	10	1
#2, #3	1343998	109255	5648	257	10	1
#1, #4, #5	1343998	109255	5180	256	10	1
#1, #4	1343998	109255	5648	257	10	1

Table 3.4: IPv4: Number of Nodes per Level with different applied rules

Applied Rules	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8
#2, #3, #5	1878824	907065	118526	12989	1339	127	16	1	-
#2, #3	1878824	907065	123961	13528	1428	141	16	1	-
#1, #4, #5	1878824	907151	118689	13003	1342	122	15	2	1
#1, #4	1878824	907151	124129	13552	1431	141	16	2	1

Table 3.5: IPv6: Number of Nodes per Level with different applied rules

Based on the above presented results we can conclude that for a typical range trie, either for IPv6 version and even more for the IPv4 version, rules 2, 3 and 5 have some minimal effect which mostly focuses on an effort to reduce the number of levels, since the resulting number of nodes is mostly unaffected (rules 2, 3 and 5 seem to have no positive effect on the number of nodes of the first level where the majority of the nodes belongs to). This observation justifies the idea and relevant experiments presented in the next section.

3.2.4. Results for Simplified Range Trie

According to the results presented in the previous section, rules 2 and 3 have minimal effect on the output range tries. However, when rules 2 or 3 (or both) are used, we need to reserve part of the memory bandwidth in order to store the shared common prefix and/or suffix. This is necessary since the shared prefix/suffix is still needed in order to be compared to the corresponding part of the incoming address and the correct branching to be determined. The amount of the memory reserved is equal to the width of the comparators, which for the CMPN mode we implemented is equal to the address width. This means 32 bits for the IPv4 version and 128 bits for the IPv6 version. Considering that the memory bandwidth is limited practically at 256 bits, it directly signifies a waste of 12.5% of memory bandwidth for the IPv4 version, while the respective waste for the IPv6 version is 50%. On the other hand, rule 5 has also small effect, but the bandwidth required for it is constant, and for the cases presented earlier only 8 bits.

Based on the above observations we decided to perform the following experiment: We remove from the heuristics completely the possibility of applying rules 2 and 3, permitting to the algorithm to apply only the rules 1 and 4. At the same time we free the reserved memory bandwidth and we make it directly available for bounds comparisons like the rest of the bandwidth. This also means that the last comparator needs to be flexible like all the rest in contrast to the original which could perform only full width comparisons. This way we hope that some additional bounds may be integrated per node which could not be integrated earlier due to the more limited available bandwidth.

Applied Rules	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5
#2, #3, #5	1343998	109255	5180	256	10	1
#2, #3	1343998	109255	5648	257	10	1
#1, #4, #5	1343998	109255	5180	256	10	1
#1, #4	1343998	109255	5648	257	10	1
#1, #4, #5, FBWU	1343998	99555	4495	254	12	1

Table 3.6: IPv4: Number of Nodes per Level with different applied rules and added Full Bandwidth Utilization

Applied Rules	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8
#2, #3, #5	1878824	907065	118526	12989	1339	127	16	1	-
#2, #3	1878824	907065	123961	13528	1428	141	16	1	-
#1, #4, #5	1878824	907151	118689	13003	1342	122	15	2	1
#1, #4	1878824	907151	124129	13552	1431	141	16	2	1
#1, #4, #5, FBWU	1878824	842579	75555	4551	262	16	1	-	-

Table 3.7: IPv6: Number of Nodes per Level with different applied rules and added Full Bandwidth Utilization

The respective results for 1,000,000 prefixes are presented in the tables 3.6 and 3.7, which constitute an expansion of the tables 3.4 and 3.5 respectively.

We observe that in the case we perform full bandwidth utilization the results are improved. Specifically, for the case of the IPv4 version, we have some quite significant decrease in the memory requirements (~ 9%) considering that the gain in memory bandwidth is only 12.5%, but in this case we do not observe any difference in the number of range trie levels. The original version had already a high branching factor at the first level, and the full bandwidth utilization allowed even higher. On the contrary, the IPv6 version displays lower decrease in the memory requirements (~ 12%) compared to the gain in memory bandwidth (50%), since the average branching factor at the first level, which contains the majority of the nodes, is very low and close to 2, in which case the bandwidth is already fully utilized (remember that when there are no long common prefixes/suffixes we have simply full bounds comparisons and the bandwidth is utilized fully in this case even in the classic range trie). However, at higher levels the branching factor increases and the full utilization of the bandwidth comes in effect. The result is that all the higher level nodes have even higher branching factor, leading to lower memory requirements and most importantly higher probability for smaller number of levels.

This phenomenon exactly we can observe it from table 3.7 where all the higher levels contain significantly fewer nodes than the previous cases and the resulting range trie is shorter for at least one level. What makes this phenomenon so intense for the IPv6 version is that we now utilize the 100% of the memory bandwidth which was previously utilized only 50% when shared common prefixes/suffixes were needed. This constitutes a 100% increase and results in the superior results we can observe in the table 3.7.

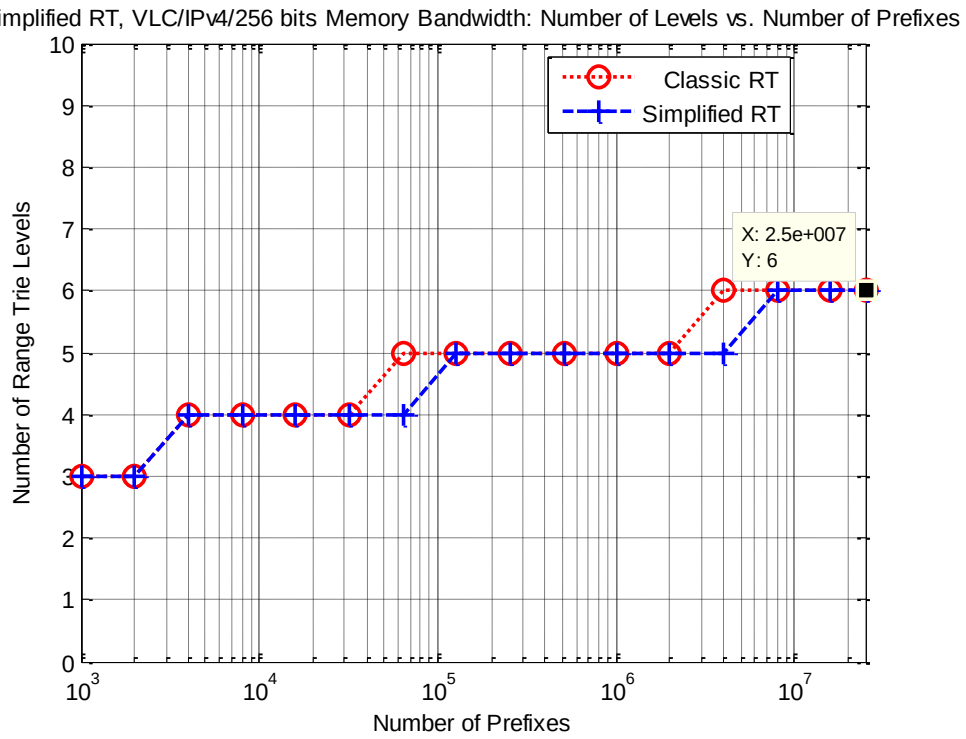

Figure 3.22: Simplified RT, VLC/IPv4/256 bits Memory Bandwidth: Number of Levels vs. Number of Prefixes

Figures 3.22 and 3.23 show the number of levels for the simplified range trie structure obtained for various sizes of lookup tables (blue dashed line) compared to the highest quality range trie structure (red dotted line) for IPv4 and IPv6 addresses respectively. In all cases we considered VLC structures with 256 bits memory bandwidth and 8 bits wide adder for address alignment. Similarly, figures 3.24 and 3.25 show the memory requirements of the simplified range trie structures as a percentage decrease compared to the highest quality range trie structure for IPv4 and IPv6 addresses respectively, while figures 3.26 and 3.27 show the average number of bounds per node.

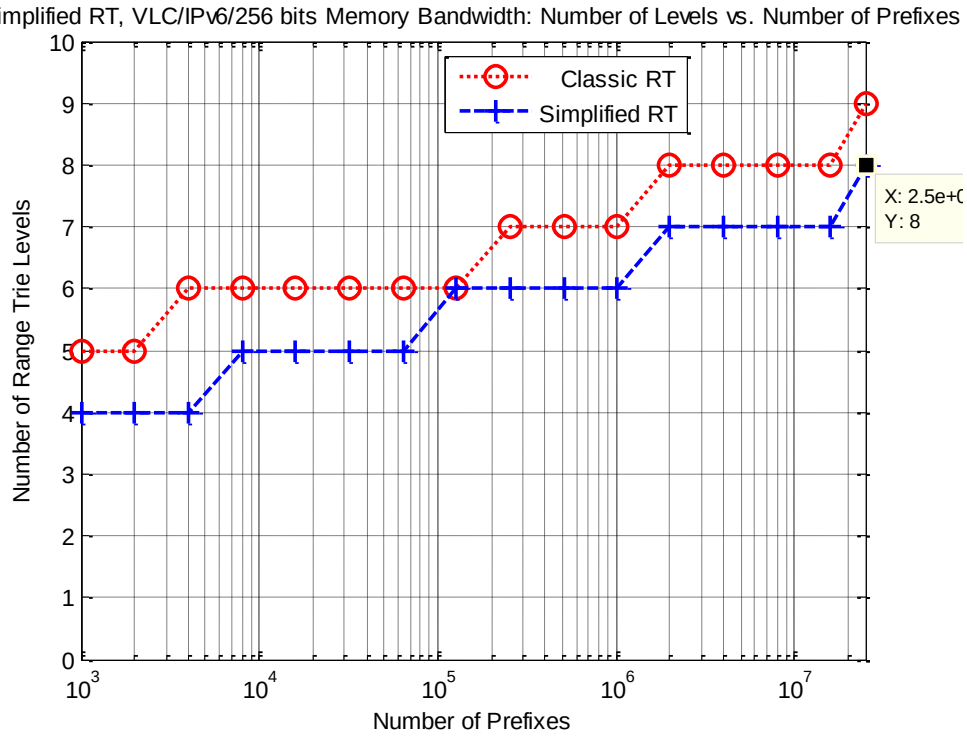

Figure 3.23: Simplified RT, VLC/IPv6/256 bits Memory Bandwidth: Number of Levels vs. Number of Prefixes

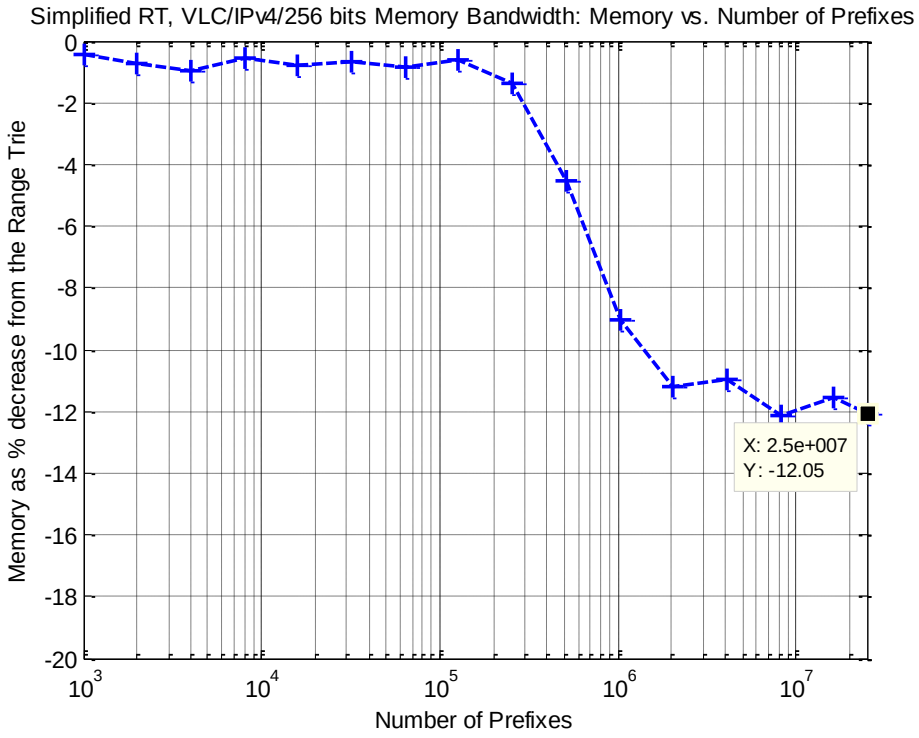

Figure 3.24: Simplified RT, VLC/IPv4/256 bits Memory Bandwidth: Memory vs. Number of Prefixes

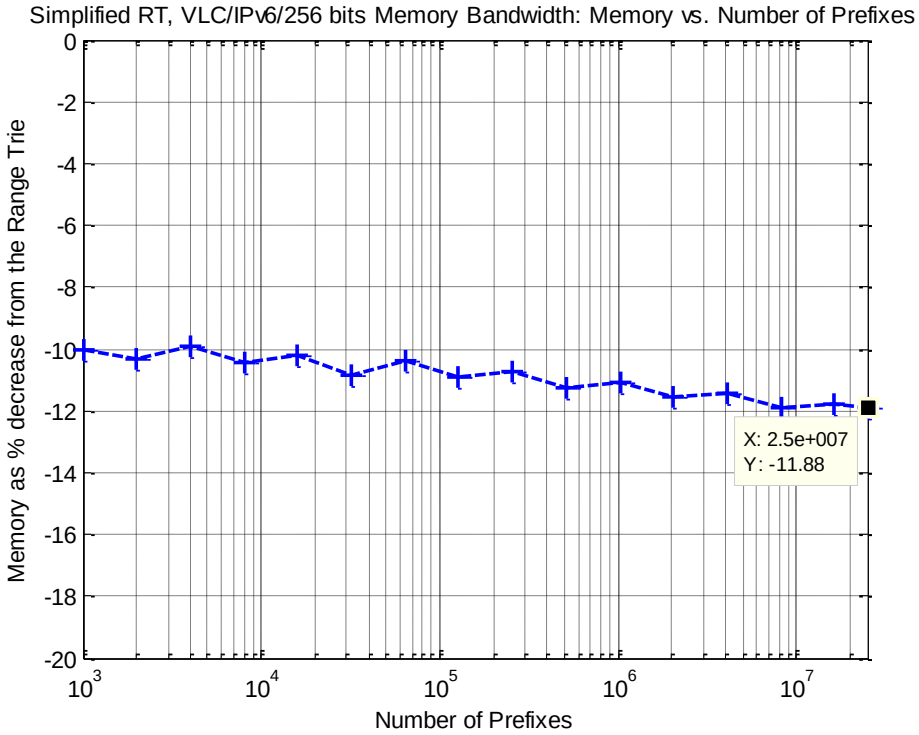

Figure 3.25: Simplified RT, VLC/IPv6/256 bits Memory Bandwidth: Memory vs. Number of Prefixes

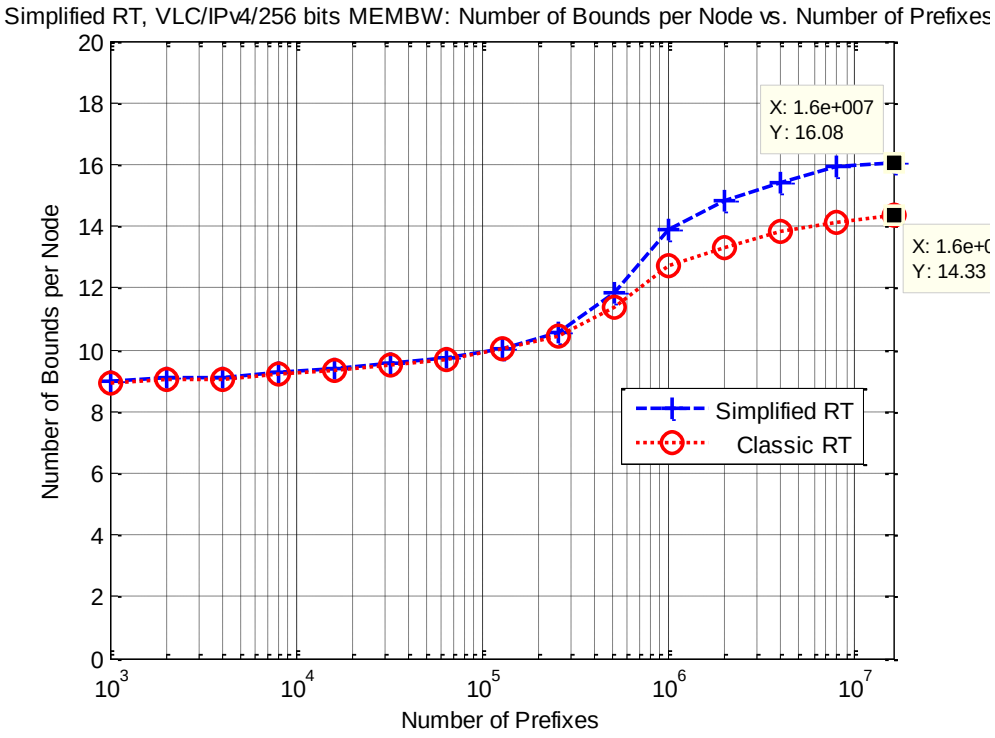

Figure 3.26: Simplified RT, VLC/IPv4/256 bits Memory Bandwidth: Number of Bounds per Node vs. Number of Prefixes

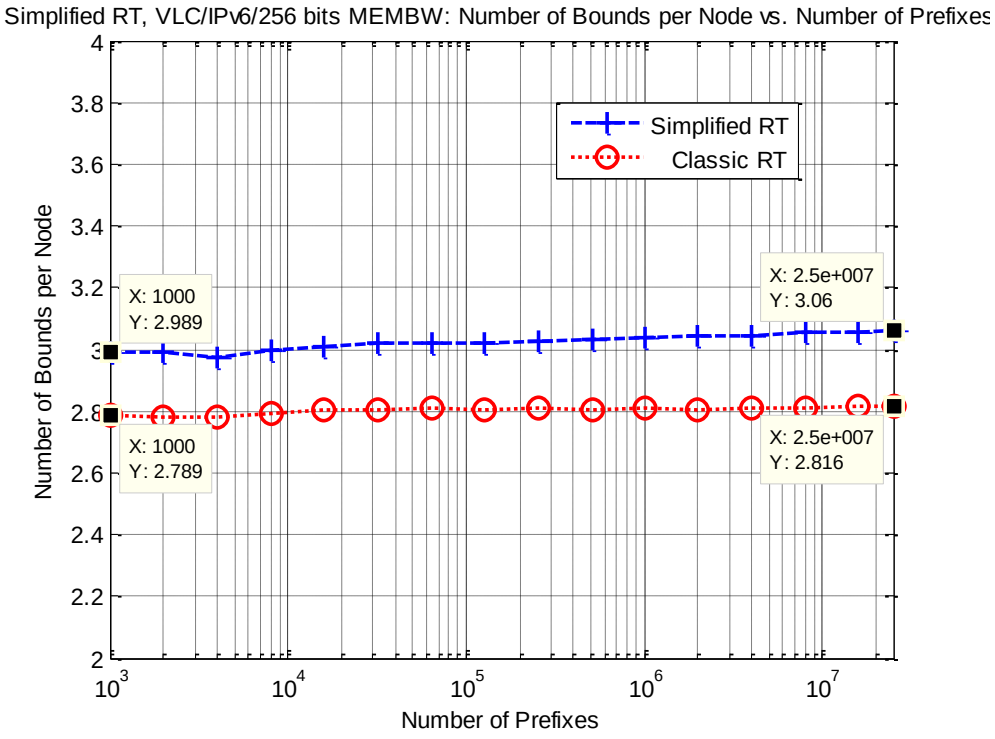

Figure 3.27: Simplified RT, VLC/IPv6/256 bits Memory Bandwidth: Number of Bounds per Node vs. Number of Prefixes

We notice that concerning the number of levels, the simplified range trie has a clear advantage over the range trie especially for the IPv6/256 bit MEMBW version. Concerning the memory requirements for the IPv6 version, the simplified range trie has consistently lower than the range trie, while for the IPv4 version we notice lower memory requirements for lookup tables of size larger than 500,000 prefixes. This behavior is justified by the fact that, for the IPv6 version, the number of bounds per node of the simplified range trie is consistently higher, while for the IPv4 version we observe a deviation in the number of bounds per node only after 500,000 prefixes, further confirming our previous analysis. More specific data concerning the simplified range trie can be found in appendix B, tables B.7 and B.8.

Based on all the previous results we can reach the conclusion that the extra utilized bandwidth of the simplified range trie seems to outweigh in all aspects the more general nature of rules 2 and 3 compared to rules 1 and 4 of the classic range trie.

This concludes the presentation of the testing performed concerning the range trie configuration generation software.

3.3. Conclusions

In this chapter we presented the software which generates the range trie configuration. A description of the software itself was presented in section 3.1, while we discussed with more details about the additions and modifications we performed on it in its subsections 3.1.1 through 3.1.4. We focused on the characteristics which good software should exhibit, along with how we managed to meet these standards concerning the correctness, reliability, versatility, flexibility, performance and quality. At the same time we also introduced certain new notions like V/SLC scheme, and provided additional details concerning parts of the algorithm which are not obvious or a straightforward application of the described optimizations in section 2.2.

Based on this final software, we performed a large range of experiments and presented the most significant results in section 3.2. In 3.2.1 we presented the quality of a highly optimized range trie structure for various implementations and a great range of sizes of the lookup tables. At the same time, we demonstrated the abilities of our software to handle very big lookup tables without any issue beyond the required time for the software, while we also noted that the execution time required increases approximately linearly to the size of the lookup tables. In section 3.2.2 we examined the effect of the VLC, SLC and V/SLC schemes on two practical implementations taking into consideration aspects like the number of levels, the memory requirements and the execution time needed by the software. In section 3.2.3 we examined the effectiveness of the rules described in section 2.2 when practically applied through the range trie construction heuristics, and based on the respective observations we experimented with a scheme which omits the second and third rule in favor of the full bandwidth utilization. The respective results were presented in section 3.2.4, from which we can conclude that it is rather preferable to rely on a simpler scheme than the full range trie as described in [6] and implemented in [4], [5] and [8], by simply removing the extra functionality of the shared common prefix/suffix, rely only on omitting the common node prefix/common zero suffix and utilize fully the available memory bandwidth.

4 HTX Platform Range Trie Implementation Support Software

For the range trie prototype implementation, it was desirable to use a system which supports an easy programming procedure of the hardware (FPGA board) and also provides to the software an efficient method of communication with the hardware, since there is a wide range of operations for which the support software has to issue specific commands to the hardware or needs to write/read data to/from it.

The HTX platform is a reconfigurable hardware computing system developed in the Technical University of Delft. This system can provide all the necessary facilities and functionality for our range trie prototype implementation as mentioned above, thus it has been selected as out implementation platform.

In the rest of this chapter we will briefly present the HTX platform in section 4.1, in section 4.2 we will describe the software/hardware communication formalism, and in section 4.3 the initialization commands generation software will be presented. Subsequently we will continue in section 4.4 with a detailed description of the range trie maintenance/verification process, which consists of reading the fixed levels, reading the spare levels, rebalancing the spare-level sub-trees and reconstructing the whole range trie.

4.1. The HTX Platform

The HTX platform consists of a standard general-purpose PC, which communicates with a Xilinx Virtex 4-100 FPGA board via high speed link (Hyper Transport bus). This board represents the reconfigurable hardware, and it is used in order to provide hardware acceleration to parts of software applications. The additional elements needed compared to a standard general-purpose PC system consist of an FPGA device driver, some few GCC 4.5 compiler extensions and a wrapper on the FPGA side [28].

The IPv4 range trie hardware design, containing three levels fixed part and three levels spare part (see section 2.2.5), has been ported in the FPGA of the HTX platform and a suitable communication wrapper for the design has been developed. A simplified schematic of the HTX platform in respect to the range trie hardware implementation can be seen in figure 4.1. Moreover, as it can be seen in figure 4.2 (source [5]), the wrapper scheme was based on queues for reasons of increased performance. More specific details about the hardware design of the range trie and about the wrapper which facilitates the communication from the FPGA side can be found in [5].

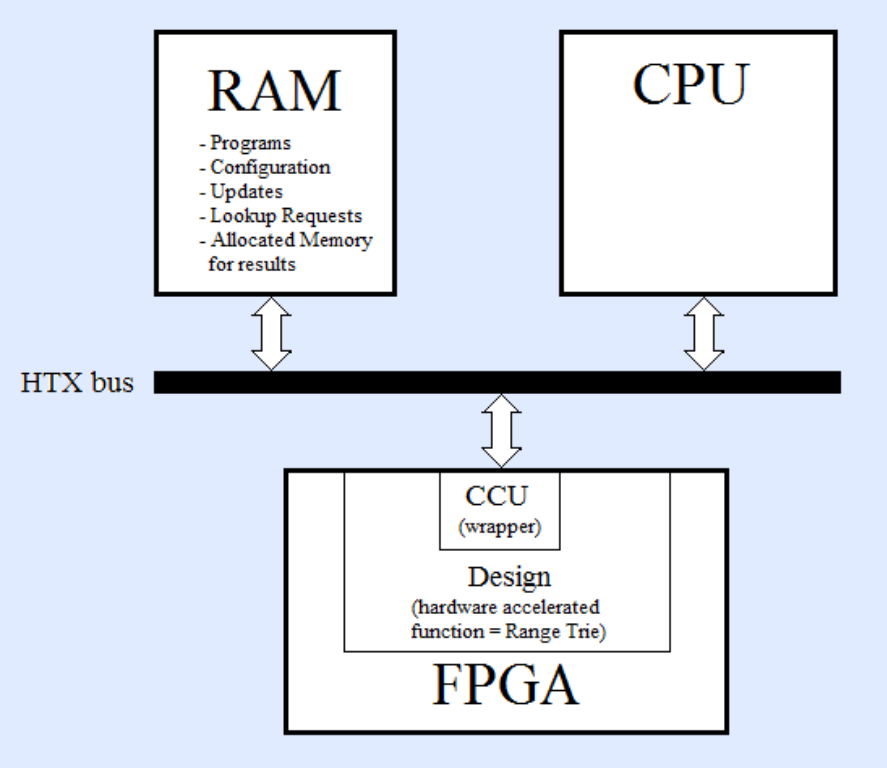

Figure 4.1: HTX Platform

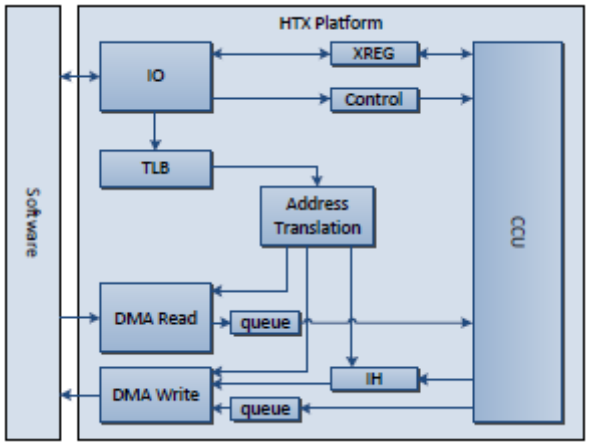

Figure 4.2: Wrapper scheme based on queues

The FPGA device driver uses system calls in order to communicate through the Hyper Transport bus with the FPGA. The modified GCC 4.5 compiler uses an additional plugin which detects in the programmer's code the function which the programmer has declared as the hardware-accelerated one, and replaces the body of the respective function with the correct system calls. These calls take care of the communication with the FPGA and all the steps needed to be executed in order to make it perform

its functionality. The FPGA device driver and the modified GCC 4.5 compiler are part of the infrastructure of the HTX platform developed in [28].

From the programmer's perspective and for the specific wrapper design, the things that need to be taken care of are:

- In each C file which contains accelerated function the header file htx-molen.h has to be included through the compiler pre-processor directive: #include "htx-molen.h"
- The function whose body is to be replaced by the plugin with system calls is denoted by the following attribute: __atribute__((user("replace")))
- The compilation has to be performed with the correct flag which will include in the compilation process the plugin, and this will replace the function with the system calls.

Due to the relatively high complexity of the shell commands to compile our C files, we generally create a makefile for it, an example of which can be seen in appendix C. In the next section we will present how we perform the communication with the tools provided to us.

4.2. Software/Hardware Communication

On the software side, the programmer needs only to allocate correctly two arrays, the first containing the data which we want to transfer to the FPGA (input), the second is created in order to be filled with the data which the FPGA returns as a response to the input (output). The word length in the HTX system, defined by the Hyper Transport bus, is 64 bits. The system main memory (RAM) is byte-addressable, which means 8 bits per address. The HTX platform demands the input array to be word-addressable, which means that the array must be aligned in memory per multiples of 64 bits (8 bytes), or equivalently the starting address for each word stored in the input array must have "000" as last three bits. This is achieved by using the function posix_memaligh() with 64 as alignment parameter.

The programmer also needs to provide the size of the input array represented in number of words (and not bytes), as well as the expected size of the output array, since the GCC compiler plugin cannot know automatically how many words it should expect until it considers that the communication operation has been completed. For the current prototype implementation of the range trie, the communication between software and hardware is always initiated from the software, which makes the above scheme ideal for use.

On the hardware side, each incoming input has to be processed, the correct actions have to be performed, and potentially data have to be returned as output. As we mentioned in the previous section, this communication functionality is implemented by a hardware module called wrapper [5].

For the overall communication, we followed a command-based protocol. Each command issued by the software starts with one word, for which some few of the MSBs (most significant bits) of this word represent the type of the command. The word length is 64 bits long, and if we take into account that this prototype design corresponds to IPv4 addresses, which means 32 bits long, the majority of the commands issued by the software can fit into only one word. However, as it will be explained later, there are two commands which require multiple words to issue.

The basic commands implemented in our design are the following:

I. no operation (NOP)
II. reset of the design memories
III. configuration of the memories with the range trie created by the software
IV. update of the range trie with a (new) prefix
V. lookup for an input address

No operation is obviously a null command, does not provide any output and does not perform any internal operation into the hardware. The reset command resets the internal memories in the design, which means that the configuration is reset. More details about what exactly the reset signal resets can be found in [5]. The configuration command fills the design memory with the configuration data. These data represent the structure of the specific range trie which has been created in the software based on the initial list of prefixes on which we want to perform the longest prefix match operation. Each configuration command represents one node of the range trie, and contains fields which convey information about all the attributes of the node. These are:

- The level of the node in the trie which, considering that each level has its own memory with its own address space, determines which memory will be enabled to store the node data
- The address of the node, which corresponds to the physical address in which the node data have to be stored
- The pointer of the node, which is a physical address which points to the leftmost child of the current node located in the next-level memory. This pointer is also used as a base pointer for accessing the action table in order to retrieve the action.
- The control data. These represent many different parameters, with most important the subtract value, the operation mode for the comparators of the design, and the common prefix length and common/zero suffix length for the node. More details about these fields can be found in section 4.4.2, while additional details and the reasons for which these choices have been made can be found in [4] and [8].
- The comparison data. These are the values which will be used for comparison with the input address. These values represent the bounds, but they are not the bounds themselves, rather a compressed version of those, as we mentioned in chapter 2.

The length of the hardware memory for comparison data (bandwidth) is 256 bits long [5], which means four 64-bis words are needed for those. Moreover, all of the remaining information needed can fit in one word, so the configuration command corresponds to a 5-words long command. The configuration command does not return any output. The update command also needs multiple words to issue. It contains the command code, a code which distinguishes low and high bound for the updating, the prefix length, the old prefix length (useful if we have prefix replacement, see [8] and [5]), the action corresponding to this prefix, as well as the low and high bound. Two words are required for this command, one for the low bound and one for the high bound. The update command returns two words with the bound and the action as a form of correctness verification. The last is the lookup command. An IPv4 address is inserted in the hardware and the longest prefix match operation is performed by the range trie. Output is a word containing the input address and the action corresponding to this address based on the longest matching prefix.

There are also some secondary commands whose role is to let the software keep track of what is the current state in the various memories in the hardware. These commands are extensively used in the work related to sections 4.4.2 and 4.4.3. We will briefly introduce them here, with emphasis mainly on the format.

I. The "Get Action" command allows us to read an entry from the action tables. Each level has its own action table, so this command contains one field representing the level/memory and one field which is the physical memory of the entry we want to read. Return value of this command is one word with the action located in that memory location.
II. The "Get Action Spare" command has exactly the same functionality, but it refers to the spare-levels part of the range trie.
III. The "Get Memory Contents" command actually reads back the configuration stored for the node in the given level/memory and physical address with the configuration command, along with the prefix lengths and action pointers corresponding to that node. For all his information ten words are required. More information about this command and relevant details can be found in section 4.4.2.
IV. The "Get Memory Contents Spare" performs the same operation for the spare-levels part of the range trie, but needs only seven words. The reason behind his will be explained in section 4.4.3.
V. Last command is the "Get Pointer Contents Spare". We need to remind here that nodes are allocated and stored in the spare-levels part dynamically, so accessing the children nodes does not happen through the node pointer, but rather through a memory which contains the physical pointers and is updated by the hardware itself with each new memory allocation. Thus this functionality is necessary in order to read the spare-levels part of the range trie.

The exact format of all the commands and the respective output can be seen in appendix D in the form of schematics and detailed description.

In the next two sections (4.3 and 4.4) we will present the software developed for this thesis related to the HTX platform support. This software utilizes the commands described above in order to initialize, maintain the range trie, and verify its correctness.

4.3. Range Trie Initialization

The configuration file generation software presented in chapter 3 produces the file which contains the description of the specific range trie configuration based on the initial lookup table. However, beyond some general parameters like the node-address width or the pointer width, the resulting data are generic and can be applied to any hardware implementation of the range trie. Thus, the file is not directly useable for configuring the HTX range trie.

As we mentioned in section 4.2, configuring the range trie in the HTX platform can be achieved through the configuration commands. For this reason a relatively simple software has been developed which reads this configuration file and generates a new file containing the corresponding HTX configuration commands in the form described in section 4.2 (see appendix D for details). Moreover, similar software reads the initial prefixes contained in the lookup table and creates the corresponding update commands. These commands are used after the configuration commands in order to fill the fixed part of the range trie with the prefix information, which means inserting prefix lengths/action pointers/actions.

We will not present more details about the software described above since it does not contain any special logic beyond a simple translation from the generic range trie representation into the HTX

commands. In contrast, the software developed and presented in section 4.4 is more complex and a deep understanding of the range trie and the address lookup process is necessary, and thus it is presented in much more detail.

4.4. Range Trie Maintenance/Verification

The structure of the range trie emphasizes in efficiency in memory requirements by compressing as many bounds as possible into the nodes and thus achieving a high branching factor and also a low number of levels despite the number of the bounds (scalability, see [6], [4] and [7]). However, this comes at the cost of increased complexity and inflexibility. From the moment the range trie representation has been created in the software and has been stored in the hardware memories (range trie configuration), the fixed part, as its name signifies, cannot be modified. No new bounds can be added or removed dynamically. Only the action pointers/prefix lengths/actions corresponding to the intervals defined by those bounds can be modified. This is amended by the use of the spare-levels part. In the current hardware implementation at least [5], the spare-levels part consists in reality a dynamic range tree structure, which allows us to add new bounds to the range trie.

However the ability of this scheme to incorporate new bounds is limited. To start with, the range tree structure does not exhibit the degree of scalability that the range trie does, since it stores only full addresses for comparison, and taking into account that the addresses are 32 bits long (IPv4) and the memory line length is 256 bits, a maximum of 8 bounds can be stored per node. In the current implementation, which contains three spare levels, the whole sub-tree under a leaf child of the last level of the fixed part can contain a maximum of 728 bounds, which is a number which can be exceeded in practice. The only way to solve this problem is by restructuring the whole range trie from scratch, so that all the bounds previously stored in the spare-levels part will be incorporated now into the fixed-levels part. However, due to the fact that a range tree created randomly by incoming bounds results to imbalances, the ability to store any new bounds will be lost long before the tree becomes full. In this case, rather than reconstructing the whole range trie, it is more efficient to just rebalance the sub-trees in the spare levels. This results in less need for total reconfiguration, and thus more time available for useful operations.

A last point of concern is that when we replace a prefix with a shorter prefix, which is equivalent to deleting the old prefix and adding a wider prefix which covers the old one, some bounds become obsolete. Whether these bounds existed in the initial configuration of the tree so they are located in the fixed-levels part, or they did not so they are located in the spare-levels part, the current implementation cannot delete bounds (though bounds in spare levels can be deleted during rebalancing, see section 4.4.4). Thus, the range trie starts getting filled with useless bounds which take valuable space from real bounds.

For all the above mentioned reasons, the range trie requires maintenance. This is provided by the software and expands beyond the limits of simple communication support, since in reality it requires an intelligent representation of the current state of the hardware range trie into the software.

In addition, since our aim was to create a prototype system design, we needed some way to see in practice what the current state of the hardware is, analyze the memory contents, and verify their correctness as well as the correctness of the software which created the configuration itself. If after

reading the memory contents of the hardware configuration we can extract the initial prefixes based on which this configuration was created, we can be sure for the correctness of the whole design (software and hardware). The support software developed and presented in this thesis fulfills this additional goal.

In the following paragraphs we will present some basic theory about the prefixes (section 4.4.1), the details about how we extract all the information from the fixed-levels part of the tree (section 4.4.2), from the spare-levels part of the tree (section 4.4.3), and how after reading the spare-levels sub-trees we can balance them (section 4.4.4) or reconstruct the range trie structure (4.4.5).

4.4.1. Prefixes Theory

Firstly we need to present some basic notions about the prefixes themselves and how we can represent them. In table 4.1 we show some example prefixes for the IPv4 case, which means that the respective addresses are 32 bits long. In the first column we have a "Prefix ID", while in the second column we can observe the way we can represent a prefix on paper, which will be called "original prefix" from now on. The '*' symbol denotes a series of "don't care" bits for the remaining bits. Thus, we say that a prefix matches to a given address as long as the bits before the "don't care" bits and the respective bits of the address are the same. We have to remind though in this point that multiple prefixes may match to a given address. The range trie structure performs the "longest prefix match" operation which means that it selects the longest (most specific, least "don't care" bits) prefix which matches to the address.

No.	Prefix - Binary	Prefix Length	Interval - Hexadecimal
A	0*	1	[00000000,80000000)
B	000*	3	[00000000,20000000)
C	011111*	6	[7C000000,80000000)
D	1011101101111000*	16	[BB780000,BB790000)
E	11111111111111111*	19	[FFFFE000,FFFFF000)
F	10101010101010101010101010*	24	[AAAAAA00,AAAAAAB00)
G	11111111111111111111111100000001	32	[FFFFFF01,FFFFFF02)

Table 4.1: Prefixes/Lookup Table Example

The third column contains the length of the prefix in number of bits. The fourth column contains the interval of the addresses which match to that prefix. Note in his point that for our purpose the interval contains the left bound but not the right bound. It is obvious that if we know this interval, we can easily determine the prefix in the form of the second column (see section 2.1). Our range trie structure uses all these bounds as comparison values to the incoming addresses in order to determine the interval in which the address belongs and through this the longest prefix (see section 2.2.3).

To clarify this with an example, let us consider the interval [00000000, 80000000). Prefixes A, B and C belong to this interval. The way the hardware works, it determines that if an address belongs to

[00000000, 20000000) then B is the longest matching prefix, for [20000000, 7C000000) is A, and for [7C000000, 80000000) is C. For the case of prefix A, we can see that the interval that corresponds to it in the range trie is different from the original interval which this prefix covers, since longer prefixes cover parts of this interval. Thus, it is obvious that from the bounds alone we cannot determine the original prefixes.

However, in the hardware we have stored information about whether a prefix corresponds to the intervals created by the bounds contained in the nodes, along with the length of that prefix. From the length of that prefix and any address belonging to that interval we can easily obtain the original prefix by keeping the first bits of the address, the number of which is determined by the prefix length, and substituting the rest with "don't care" bits. To return to our example, if we take the interval [2000000, 7C000000) and know that a prefix with length 1 corresponds to it, we can take any address belonging to it, for example 30000000, and keep only the first bit of that address and replace the rest with "don't care" bits (note that the binary representation for the hexadecimal 30000000 is 00110000000000000000000000000000). Thus, the extracted prefix will be the 0* in binary representation. This is the method we follow in our software in order to extract the original prefixes from the data stored in the hardware.

4.4.2. Reading the Fixed-levels Part

The purpose of this software code is to read the data stored in the memory banks of the fixed levels of the hardware. With the correct processing we can extract from these data the information which is stored in the hardware and as a final result we can obtain all the prefixes which are stored in any given moment.

Mode Encoding	Mode Significance	Split up of hexadecimal string "0134ABCD"
111	Not used	–
100	32	0134ABCD
011	16 – 16	0134 – ABCD
010	8 - 8 – 16	01 - 34 – ABCD
001	16 - 8 – 8	0134 - AB – CD
000	8 - 8 - 8 - 8	01 - 34 - AB – CD

Table 4.2: 32-bits comparisons splitting

As we mentioned in paragraph 4.2, the "Get Memory Contents" command returns ten words of data. The convention according to which the fields have been arranged into this command and the returned data can be seen in appendix D. First step of the prefix extraction process is to analyze the comparator modes. A total of eight 32-bis comparators are used while processing each node in the hardware. Each comparator has the following modes: full 32 bits comparison, two 16 bits comparisons, two 8 bits comparisons and one 16 bits, one 16 bits and two 8 bits comparisons, four 8 bits comparisons, or comparator is not used. The encodings for these modes are binary 100, 011, 010, 001, 000 and 111 respectively. In all cases the sum of the comparison values for these comparisons is a 32 bit long

string located in the 256-bits long comparison values memory in the hardware. These 256 bits correspond to the words two to five of the returned data (total ten words). The 32 bit string is split up accordingly to the mode. An example of this process can be seen in table 4.2. The first column shows all the possible mode encodings, the second shows how the comparator is split up in order to (possibly) perform multiple comparisons, while the third column shows the comparison values assigned to each comparison considering that the corresponding 32-bits long string is the hexadecimal number 0134ABCD.

According to the above we can determine how exactly each comparator is used. However there are two details which slightly alter the scheme presented above. Firstly, the eighth comparator can be used only if the common prefix length and common suffix length has been set as zero. For the SLC range tries this means that all of the previous comparators have been used in the 32-bits long comparison mode, but for the general case (VLC or V/SLC) this is not necessarily true (zero suffixes could potentially lead to previous comparators splitting even if the common prefix/suffix length is zero, but in our heuristics we prefer to sacrifice the last 32 bits of the bandwidth when the common prefix permits us to reduce the comparison widths). If the last comparator is used, it is in the 32-bits long comparison mode too. For this reason one single bit is enough, with 0 signifying the full 32 bits comparison, while 1 signifies that the comparator is not used. Secondly, it is possible the last *used* comparator to not be used fully. For example, it can be that the last used comparator has the mode 000 (meaning 8 - 8 - 8 - 8), but only three comparisons are performed. This is denoted by the respective not-used comparison values being zero. This is based on the fact that only the first comparison value per comparator can possibly be zero, the rest have to be different than zero. The reason for this is that the bounds are stored in the nodes in strictly increasing order. As it will become apparent from the rest of this section, a valid zero comparison value would result in violation of this rule, thus we are safe to assume that, for each comparator, any zero comparison value after the first is invalid.

Having obtained the comparison values now, we can start the process of extracting the bounds. We distinguish two cases. The first case is the common prefix length of the bounds, which we read in the control data for this node, to be zero. The common prefix is normally shared or omitted, but if its length is zero this means that we have not omitted anything. So, in this case the bounds can be extracted directly by using the comparison values calculated previously. Each comparison value produces one bound by using it as the high part of the bound and filling the rest with zeros until we complete 32 bits. For the case of table 4.2 for example, if the configuration is 8 - 8 - 16 and the common prefix length is zero, the extracted initial bounds from this comparator are 01000000, 34000000 and ABCD0000.

The second case is the common prefix length to be greater than zero, which means some part of the bound has been really shared or omitted and thus it is unknown. In that case, the common prefix has been stored in the last 32 bits of the 256 bits long configuration line (see section 3.2.4 for a possible exception if the simplified range trie scheme is implemented). The heuristics which create the configuration of the range trie are implemented in such a way that first they try to use common prefix and common suffix to reduce the width of the comparisons. Subsequently, for the VLC and V/SLC versions, a new attempt is made with the common prefix and zero suffixes (notice that it is not common zero suffix in this case). If both attempts fail, meaning that all the comparisons are still equal in size to the address length, then the last comparator can be used. In that case the software also takes care while creating the configuration to set common prefix and suffix length for the node as zero, and the same applies to the subtraction value. By doing this we ensure that whenever the common prefix and suffix are not available in the last 32 bits of the bandwidth, the prefix lengths is zero so we do not

need it. Equivalently, whenever there is non-zero prefix length, this is always available in the last 32 bits of the hardware. Note at this point that this feature has been outlined by the hardware implementation and not by the software.

However, there is a second method to determine this unknown common prefix, and in our code we use the first method only for correctness verification (if applicable, the first method as we mentioned does not work for the simplified range tries). This method is to keep track of the limits of the node. This information is not stored in the node itself, but since we can find the parent node for each node, the limits of the child node are equal to the corresponding consecutive bounds of the parent node. For example, the second child node of the parent node has node limits equal to the first and second bounds stored in of the parent node. Any address between these limits can provide us with the common prefix for the respective child node. The selection of this address in our code is the following: N^{th} child of the parent node corresponds to the $(N-1)^{th}$ bound of the parent node, while the first child corresponds to the first bound of the parent node decreased by one. From these addresses we use the first bits (equal in number to the common prefix length) as the unknown common prefix, then we append the comparison values like in the previous case, and we fill the rest with zeroes. The root node has by definition common prefix length zero, since it covers the whole address space.

For this reason, we read the nodes of the range trie top-down, in order to have already information about the bounds from the parent node available when we read the children nodes. The node pointer shows the address of the first child of that node, with the rest of the children occupying consecutive positions. Also, each node contains its address in the memory (for confirmation only, the address should be already known since it is needed to read the node from the hardware in the "Get Memory Contents" command), thus linking the intervals defined by the bounds of the parent node to the correct children nodes is relatively simple process.

The last two steps to obtain the real bounds are the following: First we add to the previous calculated values the subtraction value stored in the node, shifted accordingly to the value contained in the common suffix length field (in practice we subtract it, since the subtraction value is stored as 2's complement [8]). Second and last step is to add the shared common suffix (if any) stored in the common prefix/suffix field for a length defined in the suffix length field. In general, the process of extracting the real bounds follows the same logic as the rules that created their compressed version order. Figure 4.3 shows schematically the above described process for the general case. At this point we have extracted all the bounds and we can already make an initial verification of whether the nodes created in the software have been represented properly in the hardware.

The next step is to extract the prefixes themselves. The "Get Memory Contents" command also returns the prefix lengths and action pointers corresponding to all of the intervals of each node. As explained in section 4.4.1, using any address from the respective interval and knowing the prefix length we can calculate the prefix. In our code we use the low bound defining the interval or the high bound decreased by one for the cases that the low bound is not known (for example for the first interval of a node). The validity of the prefix is indicated by the value of the respective action pointer, with the value 00000 denoting that there is no corresponding prefix stored for this specific interval. More details about how and where the prefixes are stored in the range trie can be found in [8] and [5].

The "prefix ID" as can be seen in table 4.1 is in our case the respective action, with the difference that the action is not necessarily unique for each prefix. The action table of each level, which contains the actions, is read independently. The action pointer corresponding to each prefix is added to the node pointer and the result is the address of the respective action in the action table of that level.

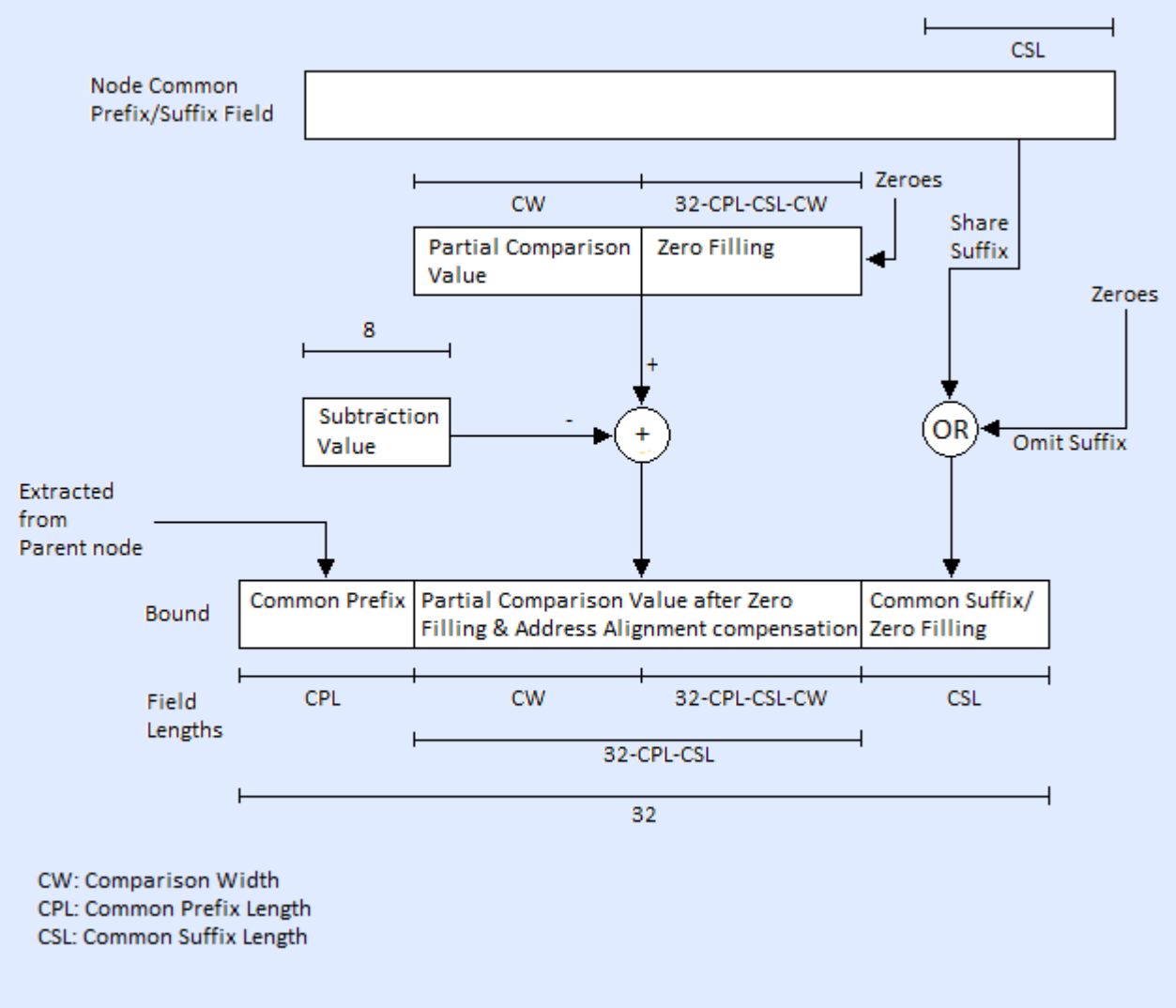

Figure 4.3: Original Bounds Extraction Process

It is possible that several prefixes have been extracted multiple times since they are located in various places in the range trie, so we perform a uniqueness check and discard multiple instances of the same prefix. Some updates may have replaced some prefixes though, so we may observe some differences. The prefix extraction at this point is complete and we should have most of the original prefixes modified by any subsequent updates. The reason we may not extract exactly the same list of prefixes is that, if an original prefix is covered completely by longer prefixes (for example 0* is completely covered by the longer prefixes 00* and 01*), it may not be stored at all within the range trie configuration of the hardware, and thus it cannot be extracted either. This would happen in the above case if all three prefixes would be stored in the same node, thus the longer prefixes “hide” the shorter in that node. If this “hiding” happens in all the relevant nodes, the prefix does not appear. Of course, this is not something negative since its existence was pointless to start with.

According to our testing, provided that the initial configuration is correct and all the initialization updates have also been provided to the range trie, our software can extract correctly the prefixes stored in the fixed part. These prefixes are stored in file for later use, see section 4.4.5.

4.4.3. Reading the Spare-levels Part

The purpose of this software code is to read the data stored in the memory banks of the spare levels of the hardware (see section 2.2.5). There are various differences from the case of reading the fixed levels, with most important being that the nodes in the spare levels are allocated dynamically by the hardware directly, in contrast to the fixed levels for which the placement of each node is defined by the software during the configuration. In the following paragraphs we will describe the equivalent process for the spare levels.

Starting point of our algorithm is to select and read a node located in the last of the fixed levels, as explained in section 4.4.2. Even if we want to read only the sub-tree corresponding to one leaf (interval defined by the bounds) of that node, we need to have the address of the node itself. Moreover, we calculate the number of bounds contained in the node from the modes of the comparators (see section 4.4.2 for more details). The number of leafs of the node is equal to the number of bounds of the node increased by one. The maximum number of bounds per node for the IPv4 version and 256 bits long comparison values memory lines is 28 (comparator mode 8 - 8 - 8 - 8 for the first seven comparators and the last is not used), which means maximum 29 leaves. From this point on, we can read the sub-tree under each and any of these leaves independently.

The first spare level contains maximum one node per leaf, which is allocated dynamically. The command “Get Pointer Contents Spare” has as input the address of the parent node and returns a memory line which contains four pointers which correspond to all of the children nodes of the parent node based on the following scheme:

The first leaf-node of the parent node is located in the address expressed by the first pointer. The seven next leaf-nodes are located in consecutive positions. Equivalently, the second pointer indicates the location of the 9th leaf-node, with the rest seven in consecutive locations, the third pointer for the 17th to 24th leaf-nodes, and the fourth pointer for the 25th to 29th leaf-nodes which is the maximum.

Figure 4.4 shows in a schematic way how exactly this pointer translation process is performed. The parent node belongs to the last of the fixed levels, while the children nodes correspond to its children nodes, all located in the first of the spare levels. The number on the left part of each node indicates its physical address in the memory.

The spare-levels nodes themselves are similar to the nodes of the fixed levels with the following differences: the comparison values are always 32 bits long with zero common prefix, which means a maximum of 8 bounds per spare-levels node and maximum 9 children. This change in reality degrades the spare levels to a range tree structure, less efficient but also not as complex as the range trie structure. In addition, the node pointer is not used anymore for accessing the children nodes or the action table, its functionality has been replaced by the stored pointers which we read with the “Get Pointer Contents Spare” command. The spare-levels nodes are read through the command “Get Memory Contents Spare” which is similar to the “Get Memory Contents” command for the fixed levels, but due to the decreased maximum number of children, the sum of the bits required for storing the action pointers and prefix lengths for a node is decreased too, which reduces the number of returned words to seven compared to the ten of the “Get Memory Contents” (see appendix D for the format of this command).

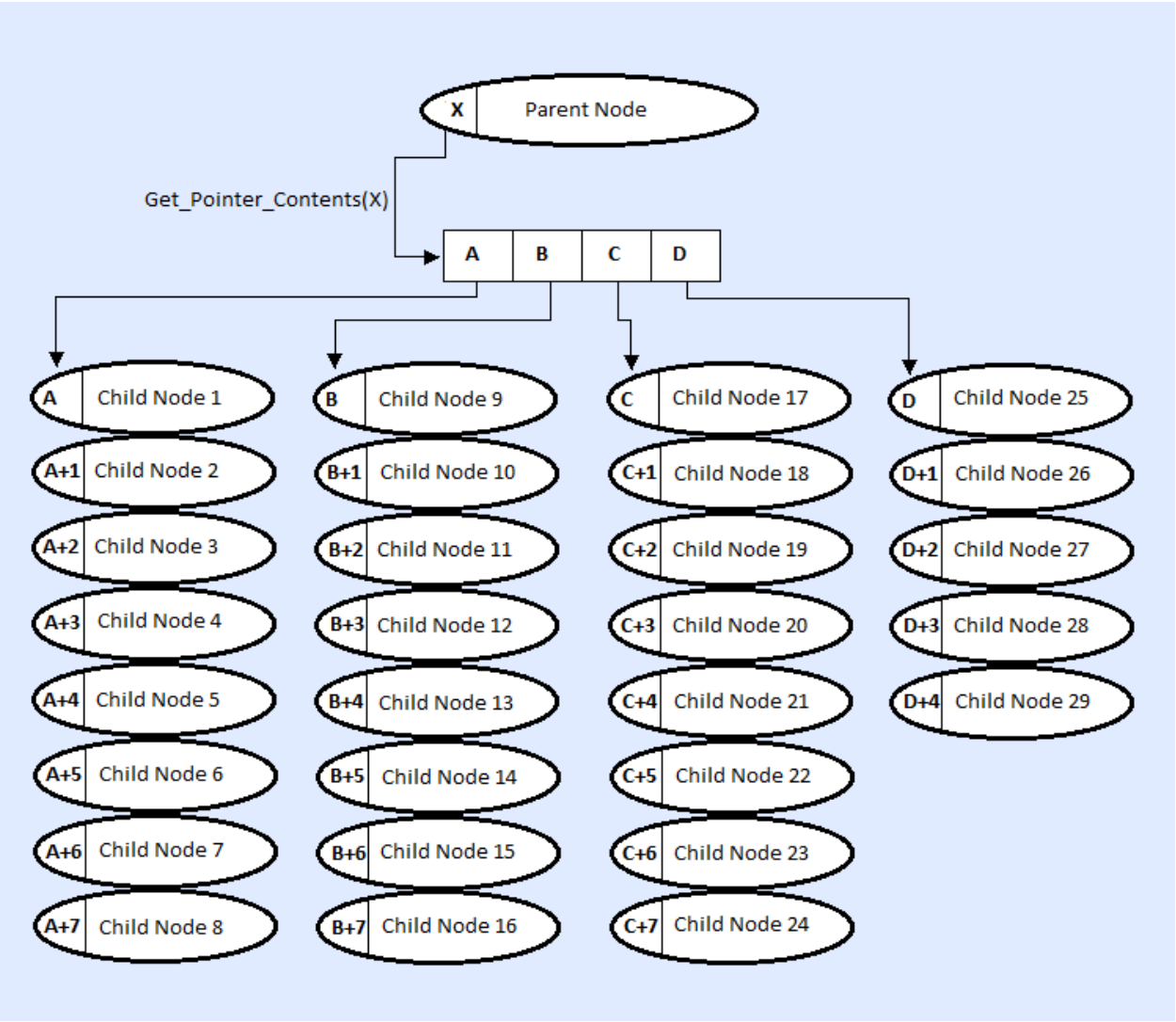

Figure 4.4: Spare part Pointer Translation

All of the control bits have preserved their format for uniformity with the fixed levels, even if they could have been changed for slightly higher memory efficiency (for example the comparator modes, common prefixes/suffixes). The address is expanded since the spare-levels memories are larger than those of the fixed part, but the node pointer field is totally missing since the hardware provides the dynamic pointers. The address of the children nodes for the next spare levels is obtained like before from the address of the parent node and the “Get Pointer Contents Spare” command with the only difference that the maximum number of children is 9 which makes the last two of the four pointers obsolete.

As mentioned before, the bounds are contained directly in the comparison values, and the prefix lengths and the action pointers are read with the “Get Memory Contents Spare” command, so the prefixes and the respective actions can be obtained exactly with the same method we followed for the fixed levels of the range trie and stored in file for later use.

4.4.4. Balancing the Spare Levels

As we mentioned before, the sub-trees stored in the spare levels have the theoretical maximum potential to store up to 728 bounds for three spare levels, but due to imbalances of the tree, it is possible that a new update cannot be inserted within the current configuration of the sub-tree. Instead of restructuring the whole range trie it is more efficient to just rebalance the specific sub-tree.

In figure 4.5 we can observe an extreme case of imbalanced tree. On the right side we can observe the order of the bounds which are provided to the range trie as updates. We can notice that the bounds are provided in increasing order. Bounds #1 - #8 are stored in the node of the first spare level. All the subsequent bounds are greater than bound #8, so they are stored to the node of the second spare level which corresponds to the last interval of the first level node. When the 16th bound is given, the node of the second level is also full. All the subsequent updates are greater than #16, so they are stored to the node of the third level which corresponds to the last interval of the second level node. When this is also full with the 24th bound, no more bounds can be stored for the last interval of the second level node, which is a problem, since bounds #25 and #26 belong to it. For this reason, before updating with #25 bound we need to rebalance the sub-tree first (or rebalance the sub-tree taking them into account too).

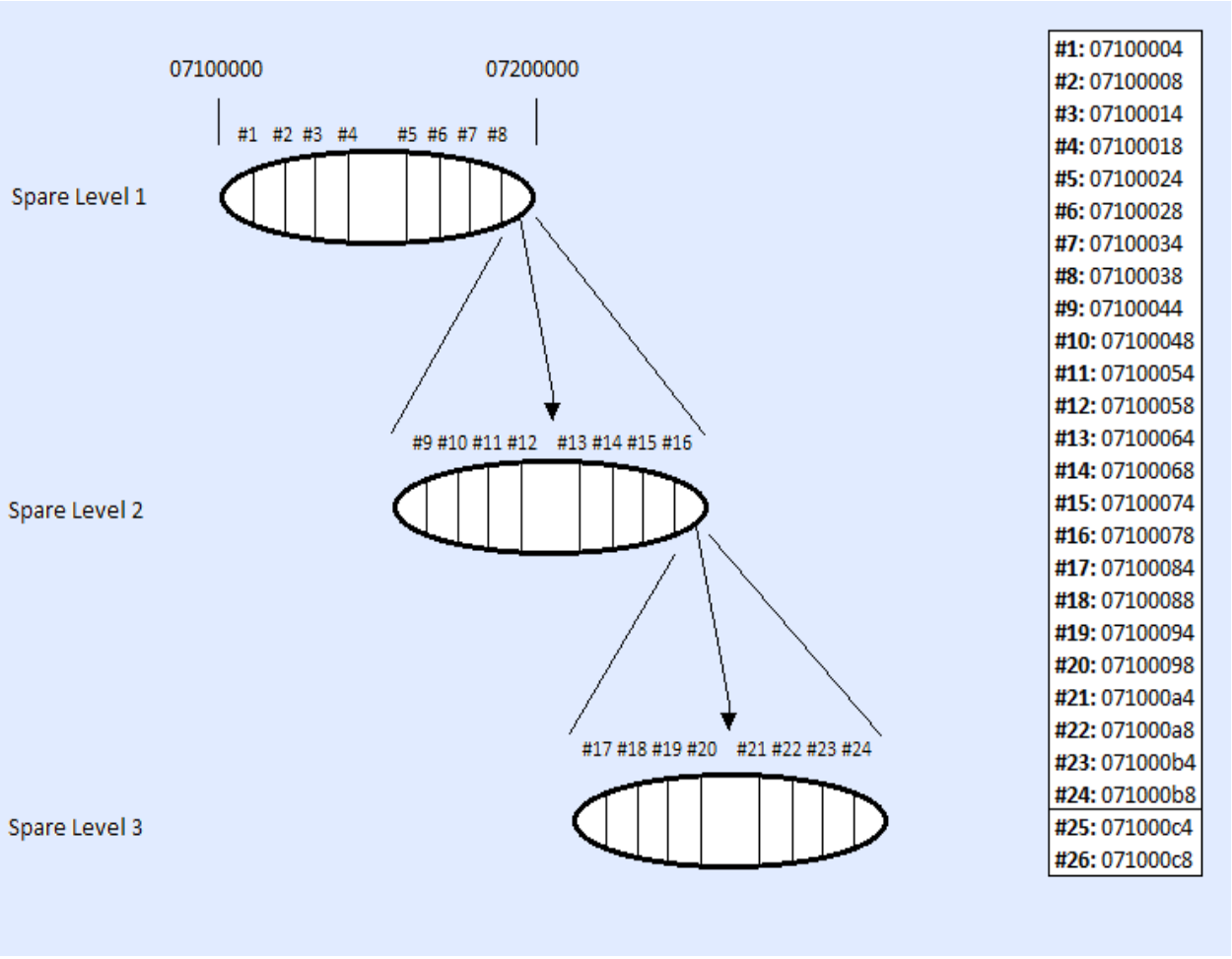

Figure 4.5: Imbalanced Spare part Sub-tree

With the procedure described in section 4.4.2, we have extracted all the prefixes stored at any given sub-tree we are interested in. If we reset the respective memories, we can recreate the sub-tree by reordering the respective updates in a way that results to a balanced sub-tree. To achieve this we split the procedure in two phases, virtual updating and real updating.

The purpose of the first phase, virtual updating, is to arrange the bounds in the sub-tree in an optimal way. The bounds can be taken directly from the prefixes we extracted from the sub-tree as described in section 4.4.2. We chose to create an as balanced sub-tree as possible, even if this leads to greater memory requirements. The reason behind this is the fact that considering that the sub-tree rebalancing process has been required, this means that the respective interval has received numerous updates, and it is safe to assume that it will receive more still. If we try to reduce the memory requirements by preserving some degree of imbalance of the tree, it is very probable that a new rebalancing will be needed very soon. For this reason we sacrifice memory in order to minimize the frequency of rebalancing and consequently the time spent for rebalancing the sub-trees compared to the normal operation time.

The scheme we follow dictates that:

- If spare level N is the last utilized for a sub-tree, all the nodes in the previous spare levels for the given sub-tree are completely full (all the nodes exist and contain 8 bounds each).
- Nodes in the Nth spare level contain a number of bounds of maximum difference one (nodes which have not been allocated/exist yet is considered to contain zero bounds).

So, each spare level is utilized to its maximum potential before a new spare level is used, and the nodes of the last spare level used have approximately the same number of bounds so that they have equal potential to accept new updates before they need rebalancing again.

The method we used to achieve this is by first ordering the bounds in increasing order, same as we do for the fixed part configuration, and then creating a reordering array which indicates the order in which we need to provide the bounds as updates to the tree. The general concept of the reordering array is to pass first, as updates, 8 bounds which separate the bounds in 9 approximately equal sets, and then recursively the same for all these groups until these sets degrade to empty sets. This reordering ensures that the resulting sub-tree will be balanced. Figure 4.6 shows the balanced version of the sub-tree of figure 4.5, after providing the additional bounds #25 and #26.

As we can notice, the rebalancing resulted in decrease of the number of spare levels used to two from three in figure 4.5, and also the nodes of the second spare level have the potential to contain six or seven additional bounds each. This way the rebalanced sub-tree can receive several more bounds until it needs rebalancing again. On the other hand, the memory requirements have increased quite a lot, since the previously 3 nodes used for 24 bounds have increased to 10 now for 26 bounds.

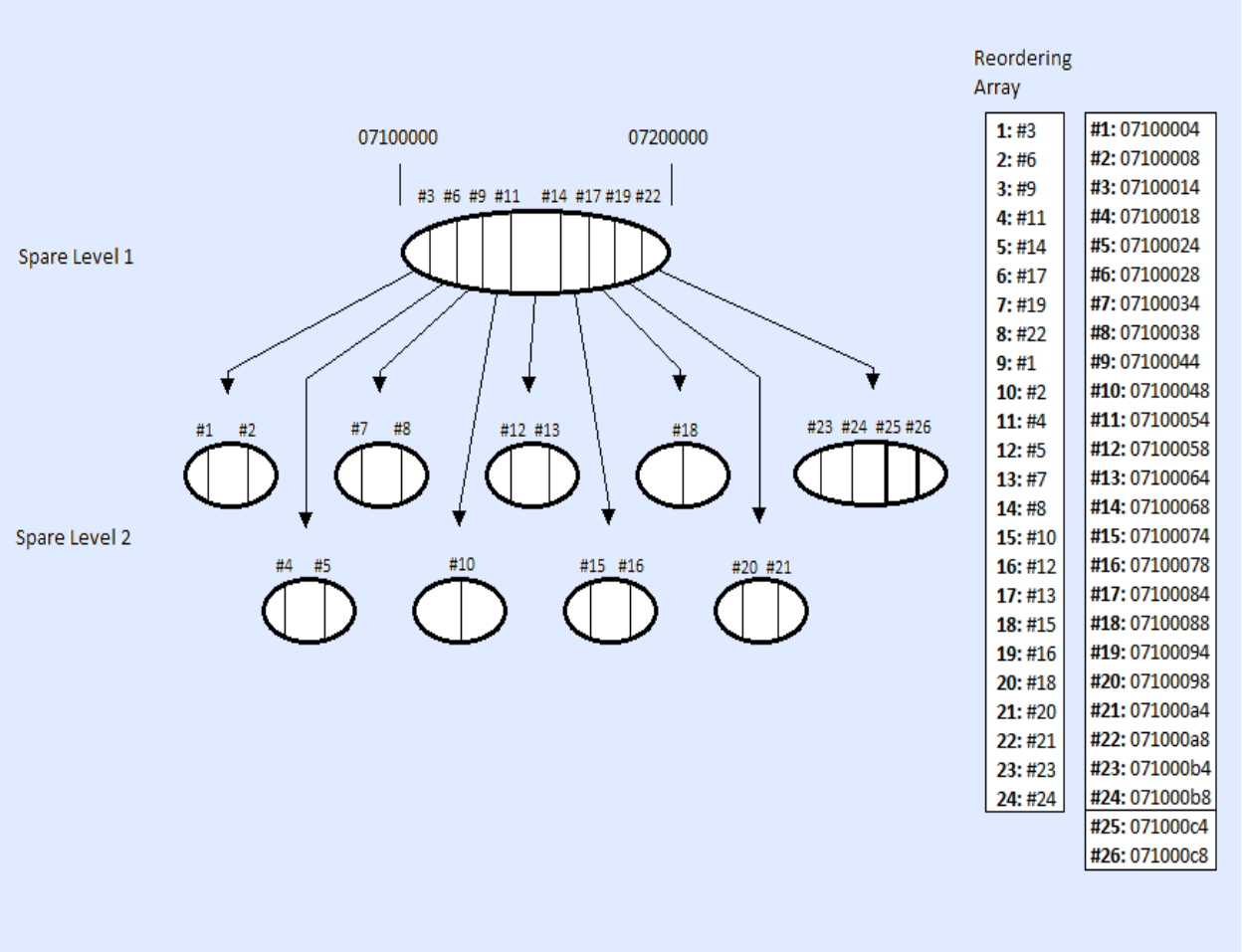

Figure 4.6: Rebalanced Spare part Sub-tree

The bounds pass as updates with zero prefix length, which enables the comparison values to be updated in the spare level nodes, but does not write anything in the prefix length/action pointers/actions field. Updates pass always bounds as couples, so the last bound has to pass as both high and low bound if the number of bounds is odd. This concludes the first phase.

The second phase is simply updating the range trie with the real prefixes, containing the real prefix length and action values. Since the bounds are already in the spare levels from the first phase, no new nodes are created, but the correct values for the prefix lengths/action pointers/actions fields are stored, creating a fully equivalent sub-tree to the initial unbalanced one.

The figures below present some important statistics concerning the rebalanced sub-trees. Figure 4.7 presents the number of spare levels required for a balanced sub-tree as a function of the number of bounds stored in the sub-tree, figure 4.8 presents the number of nodes used for a balanced sub-tree as a function of the number of bounds, while figure 4.9 presents the minimum number of additional bounds a balanced sub-tree can accept without needing additional rebalancing, considering three spare levels (worst case bound tolerance).

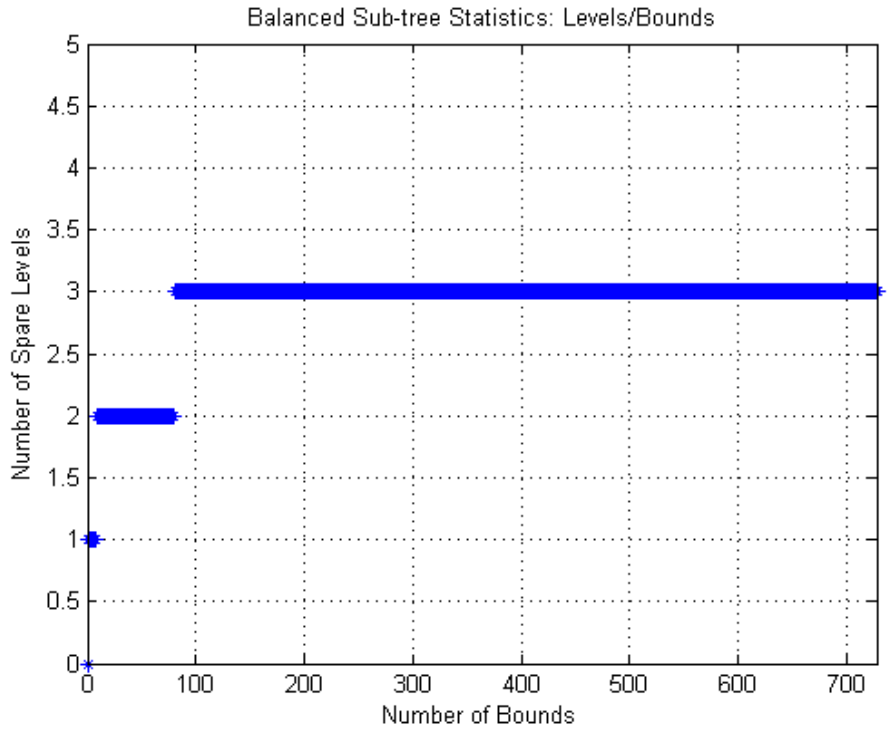

Figure 4.7: Balanced Sub-tree: Number of Levels vs. Number of Bounds

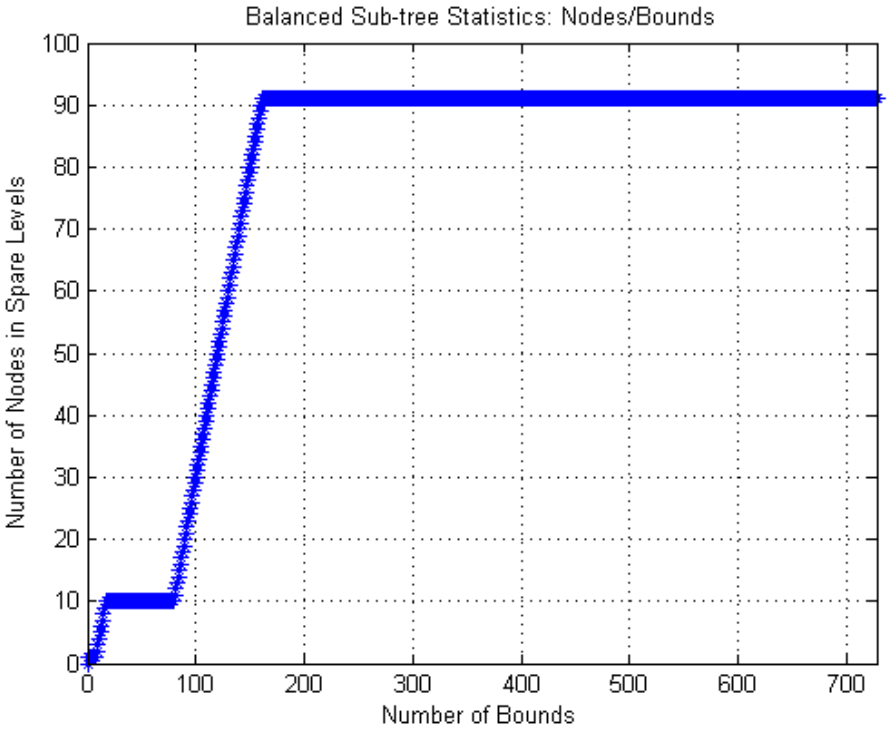

Figure 4.8: Balanced Sub-tree: Number of Nodes vs. Number of Bounds

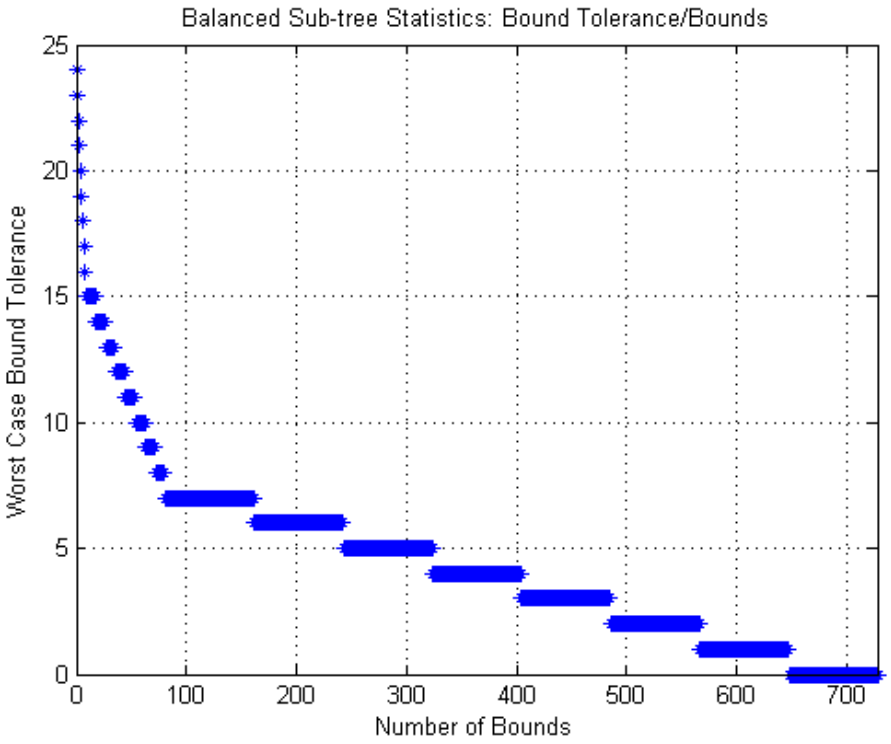

Figure 4.9: Balanced Sub-tree: Worst Case Bounds Tolerance vs. Number of Bounds

4.4.5. Total Range Trie Restructuring

After some number of updates, old bounds in the current range trie structure have become obsolete, and also the great number of updates received has increased the number of prefixes stored in the spare levels, filling the spare-levels memories with an increased number of nodes. Since the higher the number of nodes stored in the spare levels the higher the frequency of rebalancing needed, after some point the whole scheme becomes insufficient.

The method to solve this problem is to simply recreate the range trie from scratch. The prefixes of the fixed part of the range trie are extracted and stored in file according to the method presented in section 4.4.2. Moreover, section 4.4.3 describes the method of extracting the prefixes from the sub-trees stored in the spare levels. Performing this for all the sub-trees, which means for all the leaves of all the nodes of the last fixed level, results in obtaining all the prefixes stored in the spare levels. These, together with the prefixes from the fixed part represent all the prefixes stored in the tree.

Based on those prefixes, the software which creates the configuration data can be called again (see chapter 3) to create the new configuration file. This results to a totally new range trie, which has got rid of all the obsolete bounds and all the prefixes/bounds stored in the spare levels have been moved to the fixed levels, which leaves the spare levels totally empty and ready to accept new updates, ensuring greater efficiency for all the subsequent updates. However, this process demands a great amount of resources and time, thus it should not be done very frequently. The time/resources needed depends on the number of prefixes which are going to be incorporated into the range trie (see sections 3.2.1 and 3.2.2).

The resources are referred to the software, so it is of no importance to our range trie hardware as long as they are sufficient for the software to operate properly. The time can be distinguished in three parts: the time needed for reading the data from the hardware, the time needed for extracting the prefixes from the data and computing the new configuration for the hardware, and the time needed for passing the configuration to the hardware. The time needed for extracting the prefixes and computing the new configuration can be masked, since the range trie can continue operating in the hardware in parallel. When the new configuration is ready, the range trie is reset and the new configuration passes into the hardware. So, the actual time lost from the hardware perspective is the time needed for reading and writing the configuration only. A complication of this scheme is that the prefixes, based on which the new configuration of the range trie is created, stem from the instance of the range trie at the moment it was read. During the processing in the software, new updates may have been given to the range trie in the hardware. Since these updates create/replace prefixes and have not been considered in the new configuration, they have to been given as updates after the new configuration has passed successfully into the hardware. This results to non-empty spare levels when the new range trie is ready for use again, but the amount of nodes in the spare levels is expected to be only a small fraction compared to their number before the reconfiguration, while the processing time masked is the biggest portion of the total time spent.

4.5. Summary

In this chapter we presented all the work related to the HTX platform involved in this thesis. We briefly described the HTX platform in section 4.1. We continued in section 4.2 with a description of the communication protocol that the software has to adhere to in order to obtain correct communication with the HTX platform. Based on these, sections 4.3 and 4.4 describe the software developed related to initializing and maintaining/verifying the range trie on the HTX platform.

This concludes the description of the software support software developed for the HTX platform range trie implementation. In the next chapter we will present an overview of our thesis based on the work we presented and the related results.

5 Conclusions

Since its appearance Internet has developed rapidly throughout the years. Related studies and data indicate that the size of various Internet-related parameters has increased in a tremendous degree: number of Internet users, number of hosts, Internet data traffic, speed referring to both Internet bandwidth and latency, along with the requirements imposed by the users and the software utilized for various reasons (see real-time applications like video/sound streaming, instant messengers or teleconference). The current bottleneck in the Internet, which threatens to halt its further growth while maintaining the current high quality standards, is located in the forwarding engine of the routers. At each host, when a packet is received, based on the destination address contained in its header, the correct output port and the corresponding actions have to be determined by the forwarding engine through a process called "address lookup" or equivalently "longest prefix match". This consists of searching within the lookup table for the corresponding output port and actions using the destination address as "key". Due to its large and growing size and the increasing length of the addresses (IPv6 is based on 128-bits addresses in contrast to IPv4 which is based on 32-bits addresses), simple searching schemes with poor scaling to these factors are not adequate, while even more advanced ones start lagging behind.

The range trie is a new promising scheme which is based on the range trees but performs a series of optimizations which offer improved scaling capabilities. It aims at a structure with relatively low memory requirements, but more importantly facilitating a lower number of levels. Its true potential is revealed when it is combined with a hardware implementation able to take advantage of its inherent parallelism and offer very high lookup throughput and low latency [5]. Under these circumstances and especially for large sizes of lookup tables and/or address lengths its performance can be superior to other relevant techniques. The focus of this thesis and its report is on all the necessary software-related work involved in the range trie scheme.

In the rest of this section we are going to offer a summary of the thesis report in section 5.1, and we are going to continue in section 5.2 with a brief presentation of the contributions of our work concerning the range trie scheme. We are going to conclude this chapter and this thesis report with section 5.3. There we will present, based on the results obtained from our work, our suggestions about modifications/improvements along with proposals for additional future work.

5.1. Summary

This section summarizes the previous chapters of this thesis report. After a brief introduction in chapter 1 stating the address lookup/longest prefix match problem and the respective context information, chapter 2 presents all the background information concerning the address lookup

schemes. In section 2.1 and its subsections a brief description of the currently available address lookup schemes is presented with an emphasis to the algorithmic part and less to specific implementation details. Equivalently, section 2.2 and its subsections present the detailed algorithmic description of the range trie along the optimization rules constituting its basis and related structural details, while we also introduced the concept of the longest prefix matching functionality through the range tries.

Chapter 3 presents all the work related the automatic generation of the range trie configuration files. In section 3.1, the desired characteristics of the respective software have been defined, and a detailed description of the necessary modifications and additions to it has been presented subsequently. Section 3.2 continues with an extensive analysis and evaluation of the respective results which lead us to some very interesting conclusions concerning the abilities, the drawbacks and potential further improvements to the current range trie structure.

Chapter 4 contains the description of the work related to the software support of an existing range trie hardware implementation developed on the HTX reconfigurable platform. In section 4.1 we present the basic points of the HTX platform and we continue in section 4.2 with the description of the existing software/hardware interface developed for the communication with the FPGA. Subsequently, section 4.3 describes briefly the software which translates the generic range trie configuration file into the corresponding initialization HTX commands. We conclude the chapter with section 4.4, where the detailed description of the whole range trie verification/maintenance software is presented starting from an analysis about the prefixes and continuing with the description of extracting information from the fixed part, the spare part, how we rebalance the latter and how we can reconstruct the whole range trie.

The current chapter 5 offers the summary of the thesis report in section 5.1. We will continue with the listing of the contributions of our work in section 5.2 and we conclude with suggestions for improvements and additional work in section 5.3.

5.2. Contributions

In this section we will present the main contributions of our work involved with the currently presented thesis. We focused our efforts on three main points:

- Creation of a correct, reliable, versatile/flexible range trie configuration generation software with special emphasis on the high quality results and performance related details
- Use of the above mentioned software for extensive testing in respect to all the relevant parameters in order to obtain a plethora of results based on which to base our observations and conclusions.
- From scratch creation of the software which performs the software support functionality, enabling automatic handling of the HTX platform containing the range trie implementation.

In more details, concerning the first point, we performed as series of corrections which enabled us to produce correct configuration files reliably. A main advantage of our approach is that we use a single parametric code for the generation of a great variety of configuration files corresponding to different hardware implementations and/or different types of range tries by utilizing a variable set of rules. Concerning this point, we introduced an additional range trie scheme; the *variable per node/single per*

comparator comparison scheme constitutes an optimization of the *single length comparisons* scheme with superior results and no additional hardware requirements. Both of those share a reduced memory requirements encoding scheme. In addition, we introduced the *simplified range trie* which creates range tries without the help of rules 2 and 3 but taking advantage of the additional available bandwidth. Much work was involved in the process of allowing the software to handle huge lookup tables, something not feasible before, with minimal memory resources by reordering the node processing order and writing the intermediate results in files. Last but not least, certain parts of the software, including the kernel of the algorithm, have been improved from performance and/or quality of the results perspective. All the above mentioned improvements required extensive coding work. The majority of the code has been rewritten from scratch while the rest has been revised/improved with small only exceptions.

Continuing with the second point, we utilized this software to perform a series of testing. We proved that the range tries can provide a short structure (few levels) even for very big lookup tables and with relatively low memory requirements, revealing its potential and further confirming the previous results concerning this characteristic. At the same time we performed a comparative analysis of the effects of the address length, the bandwidth and the lookup table size. Subsequently, we evaluated the possible drawback of utilizing the SLC or V/SLC scheme instead of the VLC scheme in order to simplify the design and reduce the memory requirements for the encoding of the comparators. Moreover, we examined the effect and frequency of use of the rules constituting the core of the range trie optimizations. Based on the results we decided to create and test the simplified range trie which seems to perform better for the general case compared to the current range trie. In every step, wherever applicable, we offered the theoretical analysis which explains the results and which in reality constituted the base of our decision to test the effectiveness of the optimization rules and as a consequence the simplified range trie scheme.

The third main point of our work is the HTX platform support software. With this software and in combination with the work presented in [5], we provide an initial version of a functional range trie prototype implementation. The procedure until the initialization of the range trie in the HTX platform is automated by combining the range trie configuration software with the translation-to-HTX-commands software that we developed and the software/hardware interface developed in [5]. Subsequently the software can perform the standard functionality of lookups and updates, but equally important, based on software we developed, we can read the memory contents of the hardware in order to extract the range trie (or part of it) at any given moment. The process further analyzes the memory contents until the calculation of the stored prefixes and the bounds defined by them. Based on these results the range trie can be reconstructed from scratch or the spare level range trees can be rebalanced as a temporary measure.

Thus, the goals set for this thesis have been met, without this meaning that the current software is perfect or that there is no space for further improvements. Exactly this future work and suggestions concerning either the software or the general range trie structure will be discussed on the following section (5.3).

5.3. Suggestions/Future Work

In this section we will present our proposals concerning the further steps that have to be made in order to improve the range tries in general and our software in specific.

- The first step should be a further *extensive investigation of the potential of the simplified range trie scheme*, especially in comparison to the optimal alternative scheme, which is the full range trie scheme, generated with rules 2 and 3 in effect. This would provide additional evidence concerning the respective results presented in section 3.2.4, though the theoretical analysis has been performed already and supports the existing results. An option might be to integrate both in an implementation and select the fittest per node basis. This however would result in further complications and needs further investigation about whether it would be advantageous. Upon confirmation of the current results, a respective hardware prototype should be implemented at least for the IPv4 version (the current implementation presented in [5] should require only minor modifications) and further evaluation of this scheme to be performed in practice and in relation to the hardware.
- A *mechanism to detect whether the spare levels cannot support more bounds at some interval* should be integrated into the hardware and the respective information should be passed to the software. Note that the detection itself is easy but locating the sub-tree that caused this detection and suitably notifying the hardware is much harder and may need special cooperation from the software (hardware performs the detection, software inquires explicitly for the location with a new command).
- A *range trie simulator* would be a convenient addition to the existing software infrastructure. Its main purpose would be for testing and verification reasons since for practical applications other schemes would be more suitable. The reason is that the current IPv4 implementation performs up to 28 comparisons per level per cycle and normal CPUs do not have the ability to exploit properly this inherent parallelism. Moreover, even GPUs or vector processors would require multiple cycles just to unpack and align the comparison data appropriately.
- As a final step a *unified software environment* should be integrated from the existing software which will handle all the aspects of the range tries considering appropriate cooperation from the hardware. The current process is semi-automated (scripting needed) and some parts are missing (like the hardware requesting spare levels rebalancing or total reconstruction). An option would be to integrate it into the Linux kernel which would also give the advantage of being directly useable by existing applications.

This concludes chapter 5 and this thesis report. Details concerning notions presented in this report can be found in the appendices and in the respective bibliography.

Bibliography

[1] *Internet Growth Statistics*, http://www.internetworldstats.com/emarketing.htm

[2] Cisco Visual Networking Index, *Approaching the zettabyte era*, http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481374_ns827_Networking_Solutions_White_Paper.html, 2008.

[3] Internet Systems Consortium, *Internet domain survey host count*, http://www.isc.org/solutions/survey, July 2011

[4] G. Stefanakis, *Design and implementation of a range trie for address lookup*, Master Thesis, TU Delft, July 2009.

[5] Dionysius J. A. van Adrichem, *Design and Prototype of the Range-Trie IP Lookup in the HTX reconfigurable platform*, Master Thesis, TU Delft, August 2011.

[6] Ioannis Sourdis, Ruben de Smet, George Stefanakis, and Georgi N. Gaydadjiev, *Data structure, method and system for address lookup*, Patent Reference NL2002799, filed on 24th April 2009.

[7] Ruben de Smet, *Range Trie Heuristics for Variable-Size Address Region Lookup*, Master Thesis, TU Delft, July 2009.

[8] Ioannis Sourdis and Sri Harsha Katamaneni, *"Longest prefix match and updates in range tries"*, IEEE International Conference on Application-specific Systems, Architectures and Processors (Santa Monica, CA, USA), September 2011.

[9] *Classful Addressing*, http://www.freesoft.org/CIE/Course/Subnet/202.htm

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

[11] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, *Survey and taxonomy of IP address lookup algorithms*, Network, IEEE 15 (2001), no. 2, 8-23.

[12] Donald R. Morrison, *PATRICIA: Practical algorithm to retrieve information coded in alphanumeric*, J. ACM 15 (1968), no. 4, 514–534.

[13] Keith Sklower, *A tree-based packet routing table for berkeley UNIX*, Proceedings of 1991 Winter USENIX Conference (Dallas, Texas), January 21-25 1991.

[14] Pierluigi Crescenzi, Leandro Dardini, and Roberto Grossi, *IP address lookup made fast and simple*, Tech. report, 1999.

[15] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink, *Small forwarding tables for fast routing lookups*, Proceedings of the ACM SIGCOMM '97

conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '97) (New York, NY, USA), ACM, 1997, pp. 3–14.

[16] Pankaj Gupta and Nick Mckeown, *Packet classification using hierarchical intelligent cuttings*, Proceedings of Hot Interconnects VII, 1999, pp. 34–41.

[17] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner, *Scalable high-speed prefix matching*, ACM Transactions on Computer Systems 19 (2001), no. 4, 440–482.

[18] Priyank Warkhede, Subhash Suri, and George Varghese, *Multiway range trees: scalable IP lookup with fast updates*, Computer Networks 44 (2004), no. 3, 289–303.

[19] David E. Taylor, *Survey and taxonomy of packet classification techniques*, ACM Comput. Surv. 20 (2005), no. 3, 238–275.

[20] E. Spitznagel, D. Taylor, and J. Turner, *Packet classification using extended TCAMs*, Nov. 2003, pp. 120–131.

[21] S. Kaxiras and G. Keramidas, *IPStash: a set-associative memory approach for efficient IP-lookup*, vol. 2, March 2005, pp. 992–1001 vol. 2.

[22] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Commun. ACM 13 (1970), no. 7, 422–426.

[23] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese, *An improved construction for counting bloom filters*, ESA'06: Proceedings of the 14th conference on Annual European Symposium (London, UK), Springer-Verlag, 2006, pp. 684–695.

[24] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor, *Longest prefix matching using bloom filters*, Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'03) (New York, NY, USA), ACM, 2003, pp. 201–212.

[25] William N. Eatherton and Zubin D. Dittia, *Tree bitmap data structures and their use in performing lookup operations*, United States Patent 7,249,149 B1, July 2007.

[26] Jahangir Hasan and T. N. Vijaykumar, *Dynamic pipelining: making IP-lookup truly scalable*, SIGCOMM Comput. Commun. Rev. 35 (2005), no. 4, 205–216.

[27] Sailesh Kumar, Michela Becchi, Patrick Crowley, and Jonathan Turner, *CAMP: fast and efficient IP lookup architecture*, Proceedings of the 2006 ACM/IEEE symposium on Architecture for Networking and Nommunications Nystems (ANCS '06) (New York, NY, USA), ACM, 2006, pp. 51–60.

[28] Anthony Brandon, Ioannis Sourdis, Georgi N. Gaydadjiev, *"General Purpose Computing with Reconfigurable Acceleration"* fpl, pp.588-591, 2010 International Conference on Field Programmable Logic and Applications, 2010

Example Configuration File A

The following text constitutes the contents of an example configuration file for a three-level IPv4 256 bits memory bandwidth VLC range trie, with 8 bits hardware memory addresses, 11 bits memory pointers and 8 bits subtractor for address alignment. Each line starts with the letter ‘C’ and represents a node based on the following format:

C M{hardware memory level} A{hardware memory address} D{hexadecimal comparison data} D{control data} D{hardware memory pointer}

```
C M1 A00000000 D2c4c7898c000f8000000000000000000000000000000000000000000000000000
D0000000000110000111111111111111111100001100 D00000000000
C M2 A00000000 D0408100015401c252836485462406570 7b8490009c009f00a7c0af8000000000
D0100000000110100010000010100110111000110 10 D00000000000
C M2 A00000001 D2dc03000338033f03600376039c03bc03d403e404040424044484a0000000000
D0000000000110110110110110110110110100001010 D00000010100
C M2 A00000010 D344046575c68707a80949ca0ab00af80b800bc80c000cb00d000000040000000
D0100000000110000000000011011011100100011010 D00000100011
C M2 A00000011 D78007b807c307dc07f088040818085888b8f90c0930095400000000000000000
D0000000000110110110110010100111111100001001 D00000110110
C M2 A00000100 D60646c787f9094809c60a5b8c0c8cfdce600ec80f3000000000000000800000000
D0100000000110000100010000111001111100011001 D00001000100
C M2 A00000101 D000d14202c3c4148545c7078808a9096a8b4b8c4cc00d40000000000c0000000
D0100000000110000000000000000111111100011011 D00001010101
C M2 A00000110 D082e5400000000000000000000000000000000000000000000000000f0000000
D1010000000110101111111111111111111100101010 D00001101011
C M3 A00000000 D00000ca00ca83a003a8000007505e0007506b6b7b6b8000000000000000000000
D0000000000100110111001000111001111101000010 D00000000000
C M3 A00000001 D130e6000130e80003be03bf075c075e0ef573a00ef573c000000000001000000
D0000000000101001000110111001001111101000000 D00000001001
C M3 A00000010 D20000000223aecc0223aed00289b289c3000340000000000000000000000000000
D1000000000111001001000110111111111100100001 D00000010001
C M3 A00000011 D0000000030eb448030eb4500441044204ca380004ca40000507d1000507d2000
D1100000000111001001000111001001000000110000 D00000011000
C M3 A00000100 D055548ba055548bc056bf000056bf10006881b5806881b5c06b5fa0006b5fc00
D0000000000111001001001001001001000000000000 D00000100001
C M3 A00000101 D7d40b8007d40c000831813008318140 08e008e109140916000000000000000000
D1000000000111001001001000110111111100100010 D00000101001
C M3 A00000110 D6cdf78006cdf7c007f8080b4b6000000c1c91000c1c92000da800000dac00000
D0000000000101001000011001001001000010000001 D00000110001
C M3 A00000111 Da0000000aee8d300aee8d320c0000000c774a000c774c000ce20ce28d0000000
D1000000000111001001001001001001000110001000000 D00000111011
C M3 A00001000 D36cc800036cca0003780383e40000000407e5de0407e5e004180000042000000
D0100000000111001000011001001001000000010001 D00001000100
```

```
C M3 A00001001 D2d802da0390039803b003b803ea03ec040004800000000000000000010000000
D1000000000110110110110110111111111100101000 D00001001110
C M3 A00001010 D154469f015446a00182a7400182a760018800000 1885cc201885cc3000000000
D0000000000111001001001001001001001000000000 D00001011000
C M3 A00001011 D18984fc01898500018e34a0018e34b00190000000191957a0191957a819380000
D0000000000111001001001001001001000000000000 D00001011111
C M3 A00001100 D5052a000a130a140b0000000bcd0c680bcd0c700d49c0000d4a0000018000000
D1100000000110100111001001001001001001100000 D00001100111
C M3 A00001101 D16a016a8320034006c27c8006c27d0006dcd68006dcd6c009cd000009cd40000
D1100000000110110111001001001001000001100010 D00001110001
C M3 A00001110 D7ba523807ba524007c7913007c7914007db77db880000000000000000000000000
D0100000000111001001001000111001111000100010 D00001111011
C M3 A00001111 D495c4960529467005294680094009800a7a18000a7a1c000000000020000000
D1100000000110111001000111001001111001100001 D00010000010
C M3 A00010000 D00004d904d9400005bb3af005bb3af80801c7680801c77009c409c8024000000
D1100000000110111001001001001001001111001100000 D00010001010
C M3 A00010001 D12f7780012f77a001a081a103458e8003458ea00438f4390770c000077100000
D0000000000101001000111001000111000001000000 D00010010011
C M3 A00010010 D27c7340027c738002800000029451c9a29451c9c29d7984029d7986000000000
D0000000000111001001001001001001001000000000 D00010011101
C M3 A00010011 D7dc098007dc09c008000c000d3d94f00d3d94f80f180f190f4988800f4989000
D1100000000111001000111001000111000000110000 D00010100100
C M3 A00010100 D2be71ec02be71f002bff04cc2bff04d02c333a242c333a262d4436b82d4436c0
D0000000000111001001001001001001000000000000 D00010101110
C M3 A00010101 D7d635c007d6360009050905898609880c0998000c09a0000000000002c000000
D1100000000111001000111011100100111100110010 D00010110110
C M3 A00010110 D00181880488048a0bc5b9d80bc5b9e00d0f4d0f5000000000000000030000000
D1100000000110100111001000111111111100110000 D00010111110
C M3 A00010111 D3384d2163384d218338904c0338904d033c03e8033c03f000000000000000000
D0000000000111001001001001001001111100000000 D00011000111
C M3 A00011000 D3f633f643f72df403f72df804000000050d3b36050d3b3800000000030000000
D1000000000110111001001001001001111100100000 D00011001101
C M3 A00011001 D3616a7403616a780364a3000364a308036edd76e36edd77037333c6a37333c6c
D0000000000111001001001001001001000000000000 D00011010100
C M3 A00011010 D37630720376307303800000038 3eaf10383eaf183987730c3987730e00000000
D0000000000111001001001001001001001000000000 D00011011100
C M3 A00011011 D39d2f42039d2f4403a2bdb803a2bdbc03b6f28ca3b6f28cc0000000000000000
D0000000000111001001001001001001111100000000 D00011100011
C M3 A00011100 Dbc864000bc865000bcb0bcc0cef94000cef980000000000000000000030000000
D1000000000111001000111001001111111100100010 D00011101001
C M3 A00011101 D4c7b50004c7b60005a35dc005a35de006f606f68c4fec000c4fee0003d000000
D0000000000101001001001000111001001010000000 D00011101111
C M3 A00011110 D3e40dee63e40dee83eb97a583eb97a603f203f404000000000000000000000000
D0000000000111001001001000111001111100000000 D00011110111
C M3 A00011111 D188dd200188dd2801c06cd001c06ce0039a55b0039a55b806700678080000000
D1100000000111001001001001001001100011000 0 D00011111110
C M3 A00100000 D928092c094a094c0f5dc1980f5dc1a00fa618f80fa619000000000040000000
D1100000000110110111001001001001111100110000 D00100000111
C M3 A00100001 D00484c00a0394480a0394500be1f0000be1f4000c012c900c012c98044000000
D1100000000110101001001001001001001001100000 D00100001111
C M3 A00100010 D000000001a8442001a8444002568256a3a403a6040005000769000007690800 0
D1100000000111001001000110110111000001100 01 D00100011000
C M3 A00100011 D8d9780008d979000948095f8fa80fac0fc000000000000000000000048000000
D1100000000111001000010111001111111100110011 D00100100011
C M3 A00100100 D71007110a0a8b420b4300000b463c480b463c500bac01000bac011004c000000
D1100000000110110101001001001001001001100000 D00100101011
C M3 A00100101 D0040418041c046c846d0505400000000000000000000000000000000050000000
D1100000000110100110011111111111111100110110 D00100110101
C M3 A00100110 D1aa37d801aa37dc01f001f80281024e028102500400000005 1b97b2051b97b40
D1000000000111001000111001001001000000100000 D00100111101
C M3 A00100111 D55cbde0e55cbde1055e8be8055e8bec0560000005629fdf65629fdf800000000
D0000000000111001001001001001001001000000000 D00101000110
C M3 A00101000 D770bbb00770bbb8080008040805000008 0d577c080d578008bc4c6008bc4c700
D1000000000111001000111001001001000000100001 D00101001101
C M3 A00101001 D81d3540081d35800a500a580dbb61f00dbb61f80f676f678ff000000ff200000
D1100000000111001000111001000111000000110000 D00101010110
C M3 A00101010 D00000e910e9200001fb160001fb1608030a0200030a0208088008c005c000000
D1100000000110111001001001001001001100110000 D00101100000
```

```
C M3 A00101011 Dac14ac16cd426800cd427000d8d28000d8d29000dadce800dadce9005c000000
D1100000000110111001001001001001001001100001 D00101101001
C M3 A00101100 D0000000016cbf3a016cbf3c01880190040784080000000000000000060000000
D1000000000111001001000110111111111100100000 D00101110001
C M3 A00101101 D5396200053963000579600c0057960e0084008800b367c000b368000065000000
D0000000000101001001001000111001001010000000 D00101111000
C M3 A00101110 D54005800a311b000a311c000d3d18000d3d1c000dadcdae00000000067000000
D0000000000100111001001001000111111010000010 D00110000000
C M3 A00101111 D000405004927880049278900700270037053b753c94e3d40094e3d80068000000
D1100000000110101001000110111001001001100001 D00110001000
C M3 A00110000 Db2247800b2247a00b6287600b6287800d7a1c280d7a1c300edc0ede068000000
D1100000000111001001001001001001000111001100000 D00110010011
C M3 A00110001 D6be8f9906be8f9a06c0000006c6c8f406c6c8f806c7c53826c7c538400000000
D0000000000111001001001001001001001001000000000 D00110011011
C M3 A00110010 D6e0000006e333a846e333a886ec0cea66ec0cea86ef6d2246ef6d22800000000
D0000000000111001001001001001001001001000000000 D00110100010
C M3 A00110011 D2c0000002c65ec002c65ee002e00818081c00000a9cee000a9cee4006f000000
D0000000000101001001000111001001001010000000 D00110101001
C M3 A00110100 D0000000067c1cb8067c1cc006fcac0006fcae000788087708780000070000000
D1100000000111001001001001000101001001100000 D00110110001
C M3 A00110101 Db5130280b5130300c0000000c2712500c2712600cd3ccd40f8c7b400f8c7b800
D1100000000111001001001001000111000001100000 D00110111010
C M3 A00110110 D7400000074b5b6e474b5b6e6754af2f0754af2f47755f0587755f06000000000
D0000000000111001001001001001001001001000000000 D00111000011
C M3 A00110111 D00000000046e7800046e80001000aac0aac80000d151a000d151a40078000000
D1100000000111001001000111001001001001100010 D00111001010
C M3 A00111000 Db8a97400b8a97420bab02880bab028a0bacda020bacda040c000c01070000000
D1000000000111001001001001001001001111001000000 D00111010010
C M3 A00111001 D0cdc80000cdc880018df3b0018df3c004b61d0004b61d80050c050c87c000000
D1100000000111001001001001001001001111001100001 D00111011010
C M3 A00111010 D7c807cc07ef5d0007ef5d080baf45000baf46000c1212f00c12130007c000000
D1100000000110111001001001001001001001100000 D00111100010
C M3 A00111011 D7f0a54007f0a58007f8dff807f8dffc0800000008002e4c08002e4e000000000
D0000000000111001001001001001001001001000000000 D00111101010
C M3 A00111100 D111c11201545b0001545c0001ed429001ed429805053428050534300800000000
D1100000000110111001001001001001001001100000 D00111110001
C M3 A00111101 D6a806b00a2504000a2508000a5c21800a5c22000e000e20000000000800000000
D1100000000110111001001001000111111100110010 D00111111001
C M3 A00111110 D6394540063945600800000000be917b00be917b80bf12a000bf12c000f380f400
D1100000000111001001001001001001001000000110000 D01000000001
C M3 A00111111 D880000008874d8168874d81888909d6e88909d70897f87ac897f87ae00000000
D0000000000111001001001001001001001001000000000 D01000001010
C M3 A01000000 D8b9fbca08b9fbca88c6202008c6202808cf655d28cf655d48e6d2b9e8e6d2ba0
D0000000000111001001001001001001001000000000000 D01000010001
C M3 A01000001 D8f87569c8f87569e900000009074807090748080907dbdc0907dbe0000000000
D0000000000111001001001001001001001001000000000 D01000011001
C M3 A01000010 D90c1184c90c1185090e244f890e24500912c134e912c135091592a8091592b00
D0000000000111001001001001001001001000000000000 D01000100000
C M3 A01000011 D3380340037bd800037bdc000414041604594459850000000000000090000000
D1000000000110111001000110111001111001001101 D01000101000
C M3 A01000100 D55de800055df8000a37e8980a37e8a00bf40bf48e000f0a0f0a1000094000000
D1100000000111000111001000110111001001100000 D01000110001
C M3 A01000101 D98000000982186c0982187009863996698639968988 7dc889887dc8a00000000
D0000000000111001001001001001001001001000000000 D01000111011
C M3 A01000110 D9921d8009921d84099cb94c099cb94e09a6507209a650730000000000000000000
D0000000000111001001001001001001001111000000000 D01001000010
C M3 A01000111 Db5e4f400b5e4f440b86d5000b86d5400c000d000d2c12400d2c12420d4000000
D1000000000111001001001000111001000000100000 D01001001000
C M3 A01001000 D9e7ebb7c9e7ebb7e9efede869efede889f4a713e9f4a71409f6709009f670a00
D0000000000111001001001001001001000000000000 D01001010001
C M3 A01001001 D7f4718007f472000800083ed83ed200085f135d085f135d800000000800000000
D0100000000111001000111001001001111000010000 D01001011001
C M3 A01001010 D0a4ca8000a4cac001a22c0001a22c4003d003d2040000000044016d8044016e00
D1100000000111001001001000111001000000110000 D01001100000
C M3 A01001011 D484a608092a1de0092a1e000be0c9980be0c9a00c4000000c42d0a00c42d0a80
D1100000000110001001001001001001000000110000 D01001101001
C M3 A01001100 Da7180000a7c9c700a7c9c800a8000000a80d8ecca80d8ecea9068540a9068580
D0000000000111001001001001001001000000000000 D01001110100
```

```
C M3 A01001101 D97fc928097fc92a0a0597f00a0597f20adfef1c0adfef1e0c0ccce00dc000000
D10000000001110010010010010010010000100000 D01001111100
C M3 A01001110 Dae000000ae370980ae370a00ae68d800ae68e000aeb45bdcaeb45be000000000
D00000000001110010010010010010010100000000 D01010000110
C M3 A01001111 D00004d604d8000009ae510009ae51200a499f200a499f400b980ba00b0000000
D00000000001001110010010010010010011101000000 D01010001101
C M3 A01010000 D8000000095278000095280000a3221800a3222000b3c7c000b3c8c2c4b0000000
D11000000001110010010010010010010000110011010 D01010010110
C M3 A01010001 D3d9ea9003d9ea98040005000538939205389394000000000000000000b0000000
D10000000001110010001110010011111100100000 D01010011111
C M3 A01010010 Db79e8000b79e8800b7a11944b7a11946b8000000b8aeac00b8aeae0000000000
D00000000001110010010010010010010010000000000 D01010100101
C M3 A01010011 D61800a8061800b0076778000ad16c000ad16e000b620b630be000000c0000000
D11000000001110010001010010001110000011000 D01010101100
C M3 A01010100 Db25cbd40b25cbd60b8a0b8c0c000c200c3bfae80c3bfaea0c4000000b0000000
D10000000001110010001101110010010010010000 D01010110111
C M3 A01010101 D3585a0003585c0003c6b8f803c6b900057129d8057129e0080dada40bc000000
D11000000001110010010010010010010010011000 D01011000000
C M3 A01010110 D00484a0063148c8063148d00707db000707dc00074477448a44e3000a44e3100
D11000000001101010010010010001110000011000 D01011001001
C M3 A01010111 D36ef040036ef08003860b0003860c000458045c04800000000000000c0000000
D10000000001110010010010001110011110010011 D01011010100
C M3 A01011000 D400000004af916004af918007478bf00bf20c640c6800000000000000c4000000
D11000000001110010010001001110011110011001 D01011011011
C M3 A01011001 D0000040a040b0000050d5500050d5600214bc000214be00056f856f980000000
D11000000001101110010010010010010011000110001 D01011100100
C M3 A01011010 Db98a9200b98a9300c0000000d170d000d170e000d7b66000d7b66800e964e965
D10000000001110010010010010010010000100010 D01011101110
C M3 A01011011 D3c0000003c47c9c83c47c9d03e003f003f65ae883f65ae9040000000c0000000
D01000000001110010010001110010010010010000 D01011110111
C M3 A01011100 D19f6fc0019f7000025b6800025b6a0005c6062006280700000000000d0000000
D11000000001110010010010001001111110011010 D01011111111
C M3 A01011101 D2000000022fae64022fae6602780279047b4cba047b4cbc048604880d0000000
D10000000001110010010001110010011100100000 D01100001000
C M3 A01011110 D40020c004002100041604162 8b67a1008b67a1808f1a60008f1a8000d4000000
D11000000001110010001110010010010010110000 D01100010001
C M3 A01011111 D708070c0903890399cc09d00b5314960b5314980b8000000b9000000ba000000
D10000000001101101101110010010010000100000 D01100011001
C M3 A01100000 D0000000015a78e8015a78f00258c8800258c8c00400060006eff0c006eff0d00
D11000000001110010010010010001110000011000 D01100100100
C M3 A01100001 Db75b6d80b75b6e00cd98c400cd98c480d08cd08dfd88fd8c00000000dc000000
D11000000001110010010010001101111111100110000 D01100101101
C M3 A01100010 D000016a116a200005fcfee005fcfee806518651a89f0ea0089f0ec00e0000000
D11000000001101110010010001110010010010110000 D01100110101
C M3 A01100011 Daaaae000aaaae800dccd2000dccd4000eaa6eaa7edba1000edba2000e0000000
D11000000001110010010010001110010010011010 D01100111110
C M3 A01100100 De4185402e4185404e450f348e450f34ae4907978e490797ce4fe4bf6e4fe4bf8
D00000000001110010010010010010010000000000 D01101000110
C M3 A01100101 D58aa30e058aa3100637ce880637ce8c06a4574006a457480979c979ee0000000
D10000000001110010010010010010010011100100000 D01101001110
C M3 A01100110 Da2a28860a2a28880af58af60b0000000b0312000b0313000c000c600c7000000
D10000000001110010001110010010010011000100000 D01101010110
C M3 A01100111 D09152c0009152e000ad80adc67d667d8788bfc00788bfe0000000000ed000000
D00000000001010010001101110010010111101000000 D01101100000
C M3 A01101000 D8000afacafae0000c9294b00c9294b80fa34bd00fa34be00fd00fe00ec000000
D11000000001101110010010010010010011100110000 D01101101000
C M3 A01101001 Df1800000f180f50cf180f510f1c00000f213e380f213e4000000000000000000
D00000000001110010010010010010010111100000000 D01101110001
C M3 A01101010 Df3643714f3643716f384f300f384f304f4792560f47925800000000000000000
D00000000001110010010010010010010111100000000 D01101110111
C M3 A01101011 D580000005a137f005a13800060006a986aa0800000000000000000f0000000
D10000000001110010010001101111111111100100010 D01101111101
C M3 A01101100 Df88ecfc0f88ecfc2f9d95398f9d9539cfa000000fad34060fad3408000000000
D00000000001110010010010010010010010000000000 D01110000100
C M3 A01101101 Dfae67380fae67400faecc020faecc030fc7b35c4fc7b35c8fcca6a00fcca6c00
D00000000001110010010010010010010000000000000 D01110001011
C M3 A01101110 D5efbd7805efbd80067b0c00067b184408460adaefc0f5880fc0f5900fc000000
D11000000001110010010001100110010010110000 D01110010011
```

Software Testing Results

B

The following tables represent some of the results presented in section 3.2 in more detail.

Number of Prefixes	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
1K	1743	205	13	1	-	-	-
2K	3384	397	24	1	-	-	-
4K	6728	784	49	3	1	-	-
8K	13134	1505	90	4	1	-	-
16K	25729	2907	177	11	1	-	-
32K	49992	5558	319	16	1	-	-
64K	97416	10606	621	31	2	1	-
128K	189663	19889	1107	64	3	1	-
256K	367547	36954	1998	84	4	1	-
512K	712248	64603	3970	208	12	1	-
1M	1343998	109255	5180	256	10	1	-
2M	2586084	198719	11300	468	21	1	-
4M	4957551	368038	17137	908	38	3	1
8M	9462345	687564	33128	1226	65	4	1
16M	17949303	1278652	65187	2452	90	4	1
25M	27001504	1903917	78759	4058	153	7	1

Table B.1: VLC IPv4/256 bits MEMBW: Number of Nodes per Level

Number of Prefixes	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8	Level 9
1K	1972	958	124	14	2	1	-	-	-	-
2K	3920	1893	268	30	3	1	-	-	-	-
4K	7790	3771	535	60	8	2	1	-	-	-
8K	15499	7522	1007	110	13	2	1	-	-	-
16K	30917	14991	1914	212	23	2	1	-	-	-
32K	61513	29824	3839	412	44	4	1	-	-	-
64K	122325	59228	7476	827	86	9	1	-	-	-
128K	243572	118024	15318	1660	174	16	1	-	-	-
256K	485195	234871	29759	3272	338	33	4	1	-	-
512K	966201	467567	60786	6562	687	65	5	1	-	-
1M	1878824	907065	118526	12989	1339	127	16	1	-	-
2M	3742311	1809485	233491	25166	2631	253	17	2	1	-
4M	7450633	3601245	456152	50081	5150	474	57	5	1	-
8M	14835964	7166105	924542	99551	10408	1013	68	7	1	-

16M	29535886	14257606	1794100	196928	20287	1848	220	16	1	-
25M	46011777	22202713	2799910	305947	31222	3722	259	18	2	1

Table B.2: VLC IPv6/256 bits MEMBW: Number of Nodes per Level

Number of Prefixes	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5
1K	1972	229	4	1	-	-
2K	3920	460	10	1	-	-
4K	7790	916	20	1	-	-
8K	15499	1811	41	1	-	-
16K	30917	3622	76	2	1	-
32K	61513	7203	150	3	1	-
64K	122325	14298	287	5	1	-
128K	243572	28444	594	11	1	-
256K	485195	56584	1133	19	1	-
512K	966201	112425	2336	42	1	-
1M	1878824	218502	4462	76	2	1
2M	3742311	434429	8980	161	3	1
4M	7450633	864305	17678	305	6	1
8M	14835964	1717718	35499	640	12	1
16M	29535886	3415990	70096	1199	22	1
25M	46011777	5317123	107533	1965	36	1

Table B.3: VLC IPv6/1024 bits MEMBW: Number of Nodes per Level

Number of Prefixes	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
1K	1743	236	17	1	-	-	-
4K	6728	904	66	4	1	-	-
16K	25729	3449	252	16	2	1	-
64K	97416	12805	947	64	4	1	-
256K	367547	44039	3167	230	16	2	1
1M	1343998	121865	7083	276	16	2	1
4M	4957551	411419	19472	1022	44	2	1
16M	17949303	1463905	67476	3427	138	7	1

Table B.4: SLC IPv4/256 bits MEMBW: Number of Nodes per Level

Number of Prefixes	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
1K	1743	212	14	1	-	-	-
4K	6728	907	54	4	1	-	-
16K	25729	2994	202	13	1	-	-
64K	97416	10884	701	42	2	1	-

256K	367547	38000	2190	105	4	1	-
1M	1343998	115613	5832	257	10	1	-
4M	4957551	395980	18043	993	39	2	1
16M	17949303	1394877	66240	2762	102	4	1

Table B.5: V/SLC IPv4/256 bits MEMBW: Number of Nodes per Level

Number of Prefixes	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6				
1K	1972	972	173	21	3	1	-	-	-	-	-
4K	7790	3854	819	98	12	3	1	-	-	-	-
16K	30917	15250	2656	323	41	5	1	-	-	-	-
64K	122325	60295	10720	1293	162	17	1	-	-	-	-
256K	485195	239175	42371	5090	634	65	5	1	-	-	-
1M	1878824	925480	163043	19541	2435	256	17	2	1	-	-
4M	7450633	3667864	648011	77745	9690	1016	69	6	2	1	-
16M	29535886	14525586	2557703	307104	38208	4042	275	27	4	2	1

Table B.6: SLC IPv6/256 bits MEMBW: Number of Nodes per Level

V/SLC IPv6/256 (bits) Memory Bandwidth totally coincides with the table above and thus it is not repeated.

Number of Prefixes	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
1K	1743	205	12	1	-	-	-
2K	3384	397	21	1	-	-	-
4K	6728	784	42	2	1	-	-
8K	13134	1505	81	4	1	-	-
16K	25729	2906	156	8	1	-	-
32K	49992	5554	284	16	1	-	-
64K	97416	10592	547	25	1	-	-
128K	189663	19826	1048	60	3	1	-
256K	367547	36675	1769	65	4	1	-
512K	712248	62238	3283	143	6	1	-
1M	1343998	99555	4495	254	12	1	-
2M	2586084	178232	8446	292	15	1	-
4M	4957551	326611	16470	648	17	1	-
8M	9462345	606666	26466	1022	58	3	1
16M	17949303	1127869	60475	2172	64	4	1
25M	27001504	1678403	66333	2657	66	4	1

Table B.7: Simplified Range Trie VLC IPv4/256 bits MEMBW: Number of Nodes per Level

Number of Prefixes	Leaves	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8
1K	1972	887	95	6	1	-	-	-	-
2K	3920	1779	178	11	1	-	-	-	-
4K	7790	3558	363	21	1	-	-	-	-
8K	15499	7064	646	38	3	1	-	-	-
16K	30917	14013	1294	77	5	1	-	-	-
32K	61513	27791	2464	150	9	1	-	-	-
64K	122325	55234	5057	302	18	1	-	-	-
128K	243572	110065	9750	592	36	2	1	-	-
256K	485195	218505	19667	1184	67	4	1	-	-
512K	966201	434076	38831	2350	143	9	1	-	-
1M	1878824	842579	75555	4551	262	16	1	-	-
2M	3742311	1674228	148266	8887	536	32	2	1	-
4M	7450633	3326876	296942	17948	1044	63	3	1	-
8M	14835964	6607259	582948	34980	2100	126	8	1	-
16M	29535886	13118987	1163612	70421	4128	240	13	1	-
25M	46011777	20403491	1814504	107355	6264	374	23	2	1

Table B.8: Simplified Range Trie VLC IPv6/256 bits MEMBW: Number of Nodes per Level

Example “makefile” for the HTX Platform Support Software C

The following example of a makefile contains the appropriate shell commands for compiling the C code including the plugin responsible for substituting the body of the hardware implemented function with the suitable system calls.

```
GCC=/home/ce/gcc4.5/bin/gcc

PLUGIN_SOURCE_FILES= test_plugin.c

PLUGIN_OBJECT_FILES= $(patsubst %.c, %.o, $(PLUGIN_SOURCE_FILES))

GCCPLUGINS_DIR:= $(shell $(GCC) -print-file-name=plugin)

CFLAGS+= -I$(GCCPLUGINS_DIR)/include -fPIC -Wall


test_plugin.so : $(PLUGIN_OBJECT_FILES)

	$(GCC) -shared $^ -o $@


testcode: testcode.c molen_htx.c test_plugin.so

	$(GCC) -fplugin=./test_plugin.so -Wall -fdump-tree-gimple -fdump-tree-all $< molen_htx.c -o
$@

combo: testcodeSWHW.c molen_htx.c test_plugin.so

	$(GCC) -fplugin=./test_plugin.so -Wall -fdump-tree-gimple -fdump-tree-all $< molen_htx.c -o
$@


copy: copy.c molen_htx.c test_plugin.so

	$(GCC) -fplugin=./test_plugin.so -Wall -fdump-tree-gimple -fdump-tree-all $< molen_htx.c -o
$@

copy2: copy2.c molen_htx.c test_plugin.so
```

```
	$(GCC) -fplugin=./test_plugin.so -Wall -fdump-tree-gimple -fdump-tree-all $< molen_htx.c -o
$@


fixed: Fixed.c molen_htx.c test_plugin.so

	$(GCC) -fplugin=./test_plugin.so -Wall -fdump-tree-gimple -fdump-tree-all $< molen_htx.c -o
$@

updates: Updates.c molen_htx.c test_plugin.so

	$(GCC) -fplugin=./test_plugin.so -Wall -fdump-tree-gimple -fdump-tree-all $< molen_htx.c -o
$@


.PHONY : clean

clean:

	$(RM) copy copy2 combo testcode fixed updates
```

HTX Software Commands – Hardware Output D

The following table lists the HTX platform communication commands available to the software and the corresponding output of the hardware, along with a short description.

SW Command	HW Output	Description
Reset	-	Resets the hardware memories
Config [1] – [5]	-	Configures the hardware memories with the generated range trie nodes
Update [1] – [2]	Execute_Output x2	Insert or replace a prefix into the hardware, returns bounds and action
Execute	Execute_Output	Performs lookup and longest prefix match with the given address as input
Get_Mem_Contents	Mem_Contents [1] – [10]	Asks and receives all the data stored in the hardware about the specified fixed-part node, defined by its level and memory address
Get_Mem_Contents_Spare	Mem_Contents_Spare [1] – [7]	Asks and receives all the data stored in the hardware about the specified spare-part node, defined by its level and memory address
Get_Ptr_Contents_Spare	Ptr_Contents_Spare	Based on a node's address and the level of its children in the spare levels, asks and receives the dynamic pointers to their addresses
Get_Action	Action	Asks and receives the action from the defined address and action table of the fixed part
Get_Action_Spare	Action_Spare	Asks and receives the action from the defined address and action table of the spare part

Table D.1: List of Communication Commands

The figure presented in the next page shows the communication format between the hardware wrapper and the software including details about the name and significance of each field.

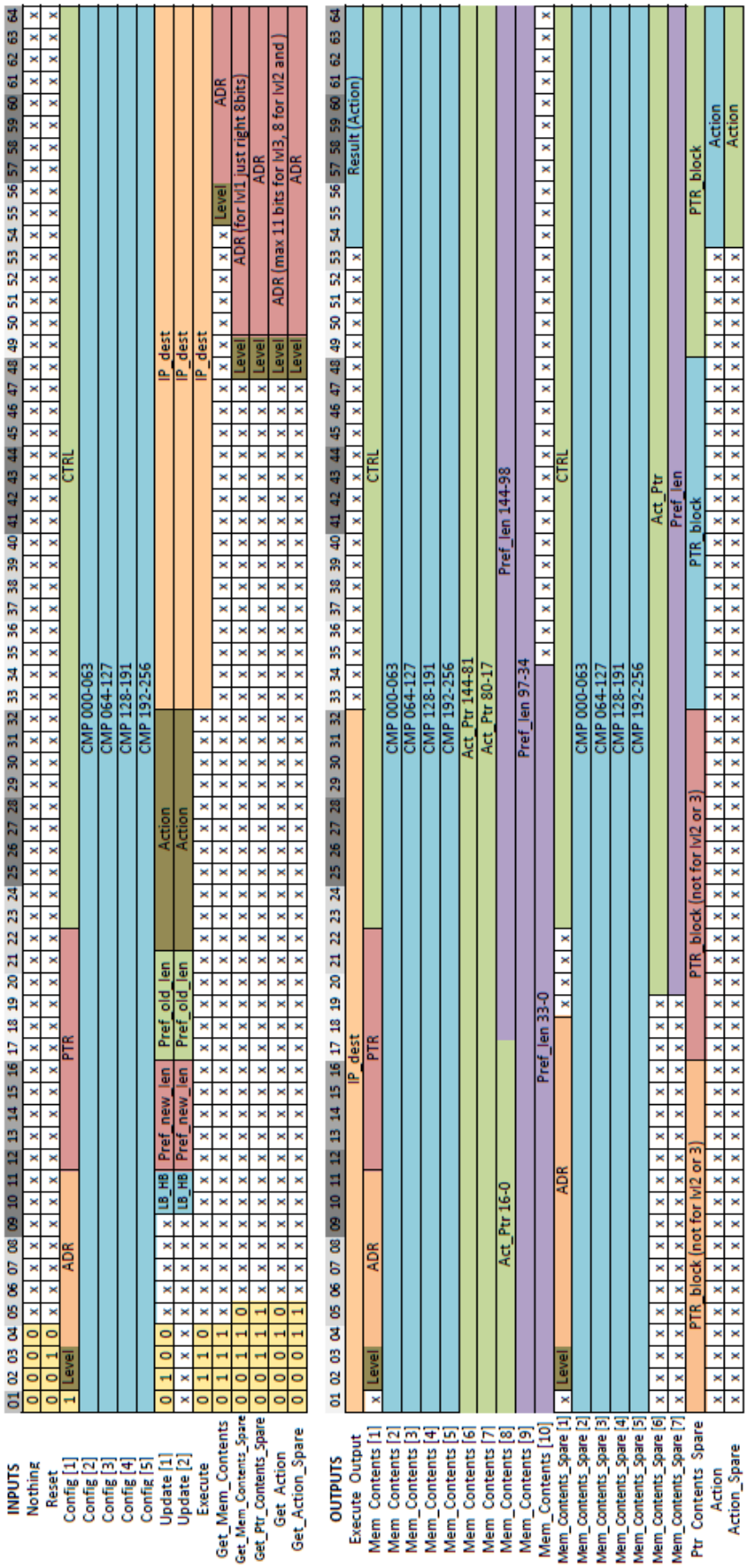

Figure D.1: Communication Commands - Format

Curriculum Vitae

Charalampos Mastrogeorgopoulos was born in Thessaloniki, Greece, on the 3rd of May in 1985. In 2003, he graduated from the 11th Unified Lyceum in Thessaloniki with GPA 19.1/20.0 (Excellent). The same year he enrolled as an undergraduate student into the Department of Electrical and Computer of the Aristotle University of Thessaloniki, Greece. During his 5-years undergraduate studies he took part in the IAESTE student exchange program in Manipal Institute of Technology, India, where he completed the project "Development and Testing of Cryptographic Algorithms". He received his diploma in November of 2008 with GPA 8.86/10.0 (Excellent), 3rd grade out of 355 students. His thesis title was "Decision Support System for Forest Fire Detection Using Optical Camera" under the supervision of Professor Loukas Petrou. At the same time, on August of 2008, he started his compulsory military service until August of 2009.

In September 2009, he enrolled as a master student into the MSc program of the Embedded Systems of the Delft University of Technology where he is a student until now. Currently, starting from October of 2011 he is a graduate employee of the Imagination Technologies Ltd., London, UK.

His research interests include, but are not limited to, embedded software, image processing/machine vision, parallel processing, embedded and real-time systems. On his spare time he enjoys the company of his friend, reading books, practicing sports, while he is a movies fan. At the same time, he always tries to keep up with the global, financial and broader technological news.