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

2011

MSc THESIS

Design and Prototype of the Range-Trie IP Lookup in the HTX reconfigurable platform

Dionysius Jacobus Adrianus (Dion) van Adrichem

CE-MS-2011-19

Abstract

In this thesis the design and FPGA implementation of the Range Trie with Longest Prefix Match and updates is completed and prototyped in an HTX reconfigurable system. The design and its implementation is an example of hardware/software co-design and supports hardware functions controlled by software using system calls. In the case of the Range Trie, the implementation is executed in hardware and controlled by software running in a Linux environment. The proposed hardware implementation of the Range Trie is compared to the Linux routing tables in terms of performance. Distinctions are made between test sets with and without updates, insertions and deletions of possible links. Various design improvements are proposed and incorporated into a Virtex-4 FPGA. For the Range Trie support in the HTX platform a suitable hardware-software interface is proposed and developed. The proposed additions and improvements lead to a functionally correct Range Trie prototype with software support for Linux. Compared to the Linux routing tables the throughput of the hardware implementation is about 47 times higher. This makes our FPGA Range Trie design a very interesting solution to the scalability problem from which the Internet and its core and edge routers are suffering.

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Design and Prototype of the Range-Trie IP Lookup in the HTX reconfigurable platform
Including interface for software support

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Dionysius Jacobus Adrianus (Dion) van Adrichem
born in Rotterdam, The Netherlands

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

Design and Prototype of the Range-Trie IP Lookup in the HTX reconfigurable platform

by Dionysius Jacobus Adrianus (Dion) van Adrichem

Abstract

In this thesis the design and FPGA implementation of the Range Trie with Longest Prefix Match and updates is completed and prototyped in an HTX reconfigurable system. The design and its implementation is an example of hardware/software co-design and supports hardware functions controlled by software using system calls. In the case of the Range Trie, the implementation is executed in hardware and controlled by software running in a Linux environment. The proposed hardware implementation of the Range Trie is compared to the Linux routing tables in terms of performance. Distinctions are made between test sets with and without updates, insertions and deletions of possible links. Various design improvements are proposed and incorporated into a Virtex-4 FPGA. For the Range Trie support in the HTX platform a suitable hardware-software interface is proposed and developed. The proposed additions and improvements lead to a functionally correct Range Trie prototype with software support for Linux. Compared to the Linux routing tables the throughput of the hardware implementation is about 47 times higher. This makes our FPGA Range Trie design a very interesting solution to the scalability problem from which the Internet and its core and edge routers are suffering.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2011-19

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 everyone close to me,
especially those who have already left...

Contents

List of Figures

List of Tables

Acknowledgements

This thesis would not have been possible without the help of many people. I would like to thank Ioannis Sourdis for the project and his guidance during the design and implementation phase. Also the other guys working on the Range Trie, Charalampos and Dimitris, thank you for the long discussions about the Range Trie, the Netherlands and Greece. I would like to thank Anthony Brandon to help getting the first programs working on the HTX platform, without these first test programs the final implementation would not have been possible.

I also would like to thank my parents and brothers and my girlfriend, Jolanda, for their support and good care during the endless testing phase and to all clients of ITCall, who kept me busy during synthesis and for their patience while I was working on the project.

Finally my gratitude goes to who helped with revising and editing to make this thesis as good as it gets.

Dion van Adrichem
Delft, The Netherlands
August 16, 2011

Introduction 1

Since the beginning of the Internet its usage and the number of connected routers and nodes has increased dramatically. Together with the number of nodes in the Internet also the traffic on the Internet will increase, see Figure 1.1. All this traffic needs to travel from the source to its destination through the Internet. At every intermediate router packets are being switched onto other links until the destination is reached. The increase in traffic and the increase in number of connecting nodes will cause problems at intermediate routers. Specifically, problems will arise at finding the correct link to forward packets to, causing the Internet to become slower.

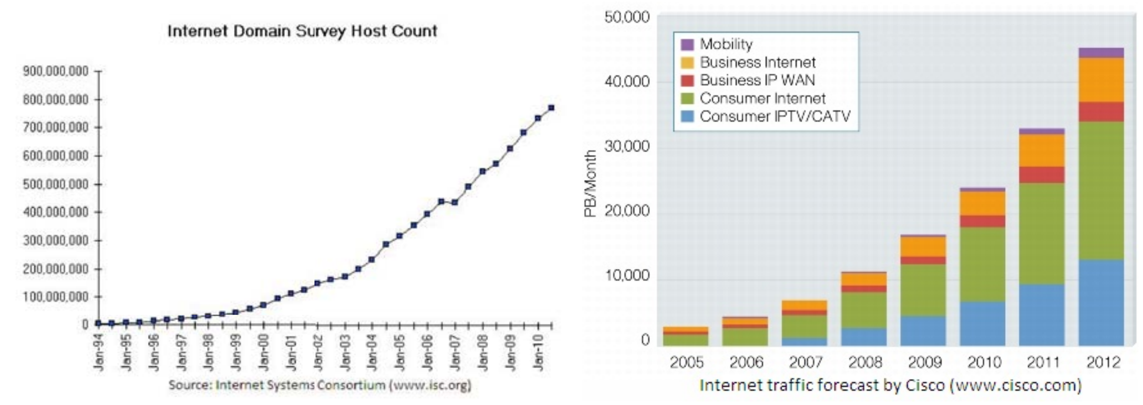


Figure 1.1: *The number of nodes as well as the amount of traffic in the Internet has increased significantly the last years [5][11]. The second figure includes the forecast by Cisco [11].*

When a network-packet is sent, it will travel through the Internet to reach its destination. As can be seen in Figure 1.2 there are routers at the interconnections of different network-links. These routers guides the packets through the network to their destination. When a packet arrives at a router the packet will be forwarded to another link depending on the destination of the specific packet.

To forward the packet to the correct output port, the router consists of two parts, the *forwarding engine* and a *switching element* as shown in Figure 1.3. The forwarding engine finds the correct output for the destination, this is also called route lookup, and the switching element forwards the packet to the link as indicated by the forwarding engine. In the forwarding engine, or *routing table*, IP address ranges are stored. These address ranges cover the complete IP address space as shown in Figure 1.4.

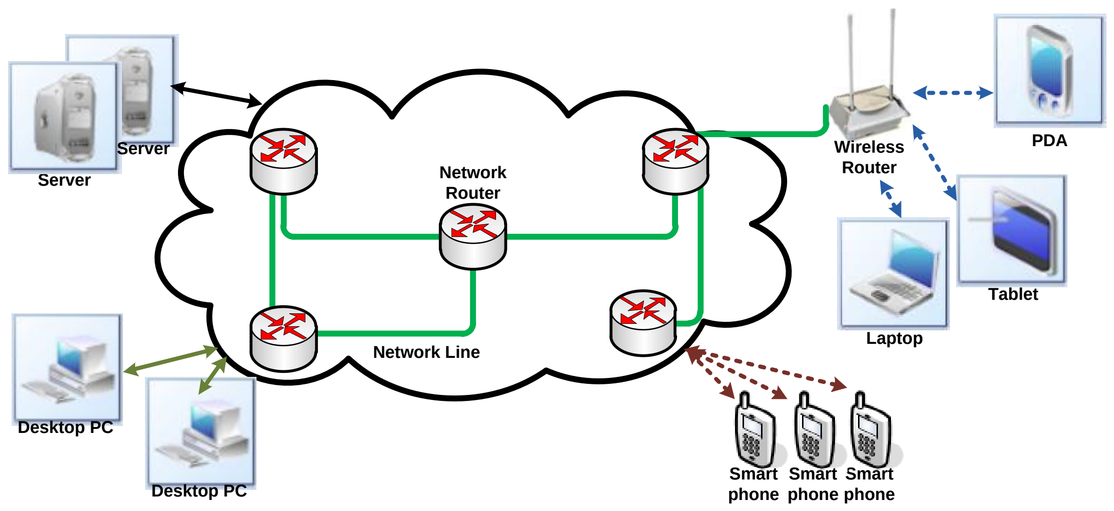


Figure 1.2: *Example of a network where different types of appliances (nodes) are connected to the network. Routers allow the communication between the nodes by forwarding packets from incoming to the correct outgoing links. Image from [21]*

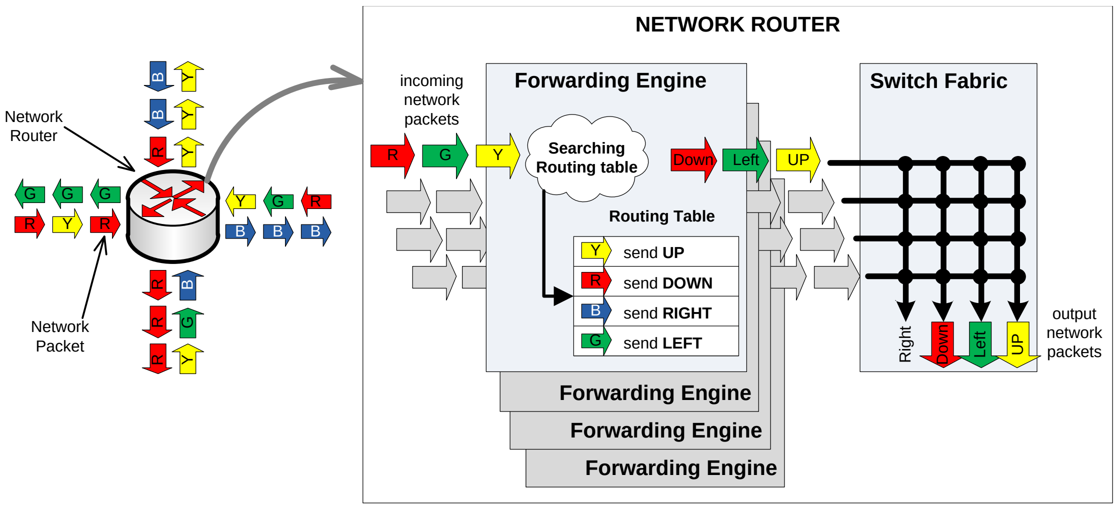


Figure 1.3: *A router consists of two parts, a forwarding engine which checks to which output port an incoming packet should be forwarded to, and a switching element which does the actual forwarding of the packet to the specified port. Image from [21]*

The address ranges are stored as *prefixes*, for example 1101xxxx as 8 bits IP address. If a routing table contains an entry with the prefix 1101xxxx, this means that the forwarding engine knows a correct output port for destination addresses 11010000 to and including 11011111. To differentiate between different address ranges also the length of the range is stored used in the notation. IPv4 IP address ranges can then be written as 192.168.0.0/16 which means that the binary representation of 192.168 is the prefix and the range starts at 192.168.0.0 and end at 192.168.255.255.

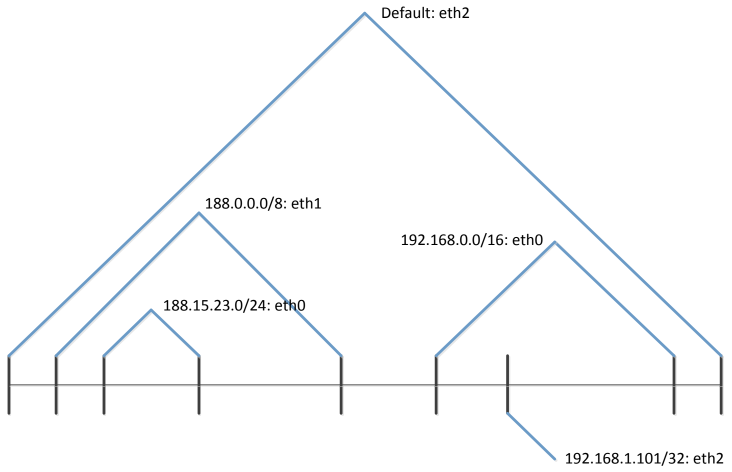
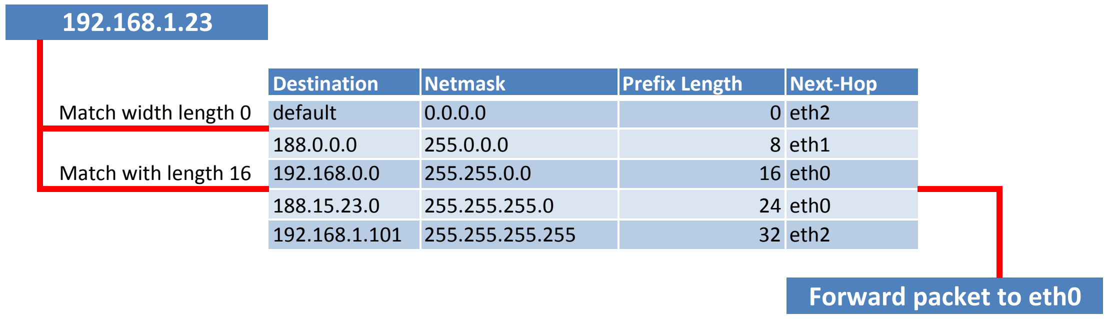


Figure 1.4: *The complete address space can be seen as one large range, every route that is added to the routing table can be seen as a new range with the new output port associated.*


192.168.1.23
Match width length 0
Match with length 16
Forward packet to eth0

Destination	Netmask	Prefix Length	Next-Hop
default	0.0.0.0	0	eth2
188.0.0.0	255.0.0.0	8	eth1
192.168.0.0	255.255.0.0	16	eth0
188.15.23.0	255.255.255.0	24	eth0
192.168.1.101	255.255.255.255	32	eth2

Figure 1.5: *A routing table is used to determine the correct output port for an incoming packet. A lookup on the destination address has to be performed and the correct next hop or output device has to be selected using a Longest Prefix Match. In this case eth0 is the correct output port as the selected route has a longer matching prefix than then the default route which matches all addresses.*

In Figures 1.4 and 1.5 the prefix/prefix-length notation is also used. Both figures give a clear view of how a routing table can be seen as list of prefixes and ranges in the IP address space. The lookup of a certain destination address can be seen as finding out which address range with largest prefix contains the destination address. When the range with the longest matching prefix is found, also the correct output port is found and the switching element can forward the packet to the correct port.

When the routing table stores the ranges as prefixes, then a lookup will be: finding the longest matching prefix or Longest Prefix Match. As an address 10110011 matches both prefix 1011xxxx and 10xxxxxx it is up to the lookup mechanism to select 1011xxxx as longest matching prefix.

With the future increase of the Internet in mind IPv4 needs replacement by a newer version which supports more connected hosts, IPv6. To support more connected nodes

IPv6 has a larger IP address containing 128 bits instead of the 32 bits used in IPv4. This has major consequences for the routing tables and their address lookup algorithms, in terms of memory usage as well as performance as the current algorithms do not scale well.

This introduction chapter deals with the problem definition and goals which form the basis for this thesis in Section 1.1. In Section 1.2 there is a description of the goals for this thesis and the associated contributions. An overview of the remaining thesis is given in Section 1.3.

1.1 Problem

Due to the increase of the Internet in terms of number of hosts and in terms of traffic and throughput, routing tables have increased in the number of entries. Also the IP addresses will increase in size due to the transition from IPv4 to IPv6. As current IP Lookup algorithms do not scale well, to address width and to number of routing table entries, in terms of performance and cost as well as in power consumption and silicon area. This will increase the latency for a single lookup and the total end-to-end delay per packet. This thesis will investigate a new method for IP lookup and its performance is measured while implemented in an FPGA and prototyped in the HTX Reconfigurable Platform.

The current methods to store prefixes and perform lookups on routing tables do not scale well with the number of entries and with an increase in address width. As both variables are about to increase significantly this causes that each lookup will get a larger delay than currently is expected. This inevitably means that the Internet on itself will become slower.

Before a packet reaches its destination, it will be forward by multiple routers. Each intermediate router will have to perform an address lookup to determine the correct output port for this specific packet. Whenever this lookup will consume more time, then also the total end-to-end delay for each packet will increase. With the increasing traffic and faster connections a larger end-to-end delay is not acceptable.

Next to the problem with performance, there also is a problem with costs. As larger routing tables lead to larger memories which consume more power and resources. The lookup, of an address itself, will become more complex due to the larger addresses which also will lead to more complex functions which require more resources.

The problem is at the forwarding engines in routers. More specifically, the way current mechanisms maintain the routing table and perform a lookup contribute to the bottleneck resulting in decreased performance and increased costs. A possible solution lies in new methods and much research is currently being performed in this area. Instead of changing the algorithm it is also possible to speed up a specific method by implementing it in hardware or in an FPGA. This can give quite some speedup compared to software, depending on the algorithm.

The Range Trie is one of the route lookup mechanisms that can be the solution of the increased delay as it does many optimizations to minimize the number of memory accesses and to minimize the complexity of the lookup itself. As opposed to the current most used mechanisms such as the Radix Tree [1] and LC-Trie [15], which do optimizations though less than the Range Trie.

This thesis will explore the Range Trie in combination with an FPGA implementation and whether it is interesting for further investigation and possible implementation in a router. To investigate the behavior of the Range Trie in an FPGA it will be connected to an HTX enabled computer. In such that the Range Trie hardware implementation can controlled and observed from software via the HyperTransport Bus [4].

1.2 Thesis Objectives

As said in the previous section problems arise with the current routing table implementations. This thesis will explore if it is possible to use the Range Trie in an hardware implementation and whether it is a field of interest for a final solution to the problem.

The main goal of this thesis is to deliver an FPGA implementation of the Range Trie Route Lookup working in an HTX reconfigurable machine. With this implementation the speed and costs of lookups will be compared to the default Linux kernel lookup method to determine if the hardware accelerated version of the Range Trie is a field of interest as replacement for current routers.

The objectives of this thesis are:

- **complete the Range Trie hardware design which supports LPM and Updates**. Currently, there already is a base design with most of the components as delivered by [12]. This base design has to be tested and adjusted where it is not fully functional.

- **FPGA prototype the fully functional Range Trie hardware design**. As the design uses a relative large amount of memories which are in the base design implemented in LUTs this might become a problem. These memories might have to be replaced with BRAMs in order to fit the design on an FPGA.

- **port the FPGA design to the HTX platform which offers reconfigurable acceleration with software support**. There are some demands by the HTX platform on the accelerated function. The FPGA prototype design has to meet the demands of the platform in order to run successfully.

- **evaluate the efficiency of the final design when ported in an FPGA in terms of cost/performance compared to software**.

1.3 Thesis Overview

The structure of this thesis is ordered such that the reader is taken along the design of the Range Trie and its implementation in hardware including the testing and evaluation. Therefore chapter 2 contains all required background information about the routing tables, some of its implementations, including the Range Trie, and the hardware platform on which the Range Trie will be implemented.

With all required information it is possible to continue with the design and implementation of the Range Trie. This is discussed in chapter 3. It includes all required modifications on the initial Range Trie design. These modifications include the additions for correct interfacing to the hardware platform as well as additions needed by the software to retrieve all the required information. After the design is completed it can act as routing table for a Linux computer. This gives the possibility to make a comparison to the default Linux lookup methods in terms of performance. In chapter 4 the performance of the implementation is measured with multiple scenarios to highlight different aspects of the Range Trie implementation.

In chapter 5 the results of the measurements will be concluded as well as the implementation of the final design. Next to an overall summary it also contains some possible improvements and future work is proposed for the Range Trie and its implementation.

Background 2

The main goal of this thesis is to implement the Range Trie on a general purpose processing platform with support for hardware acceleration of functions and compare this with a software based routing table.

In this chapter, the required background will be given, starting in Section 2.1 with routers and how they work with routing tables. Three different IP Lookup algorithms are discussed in Section 2.2. The Range Trie algorithm will be compared to the other algorithms in Section 2.3 to give the reader a feeling of the amount of optimizations the Range Trie uses and how it works for lookups and updates. After this, in Section 2.4, the platform on which the Range Trie will be implemented will be discussed.

2.1 Routers, Routing and Lookup Tables

Routers are the intersections of networks, at these intersections multiple links come together and packets will be transferred from one link to another in order to let the packets reach their destination. As shown in Figure 2.1, a router receives packets from any of its incoming interfaces. With each of the received packets a lookup based on its destination address is performed in the forwarding engine.

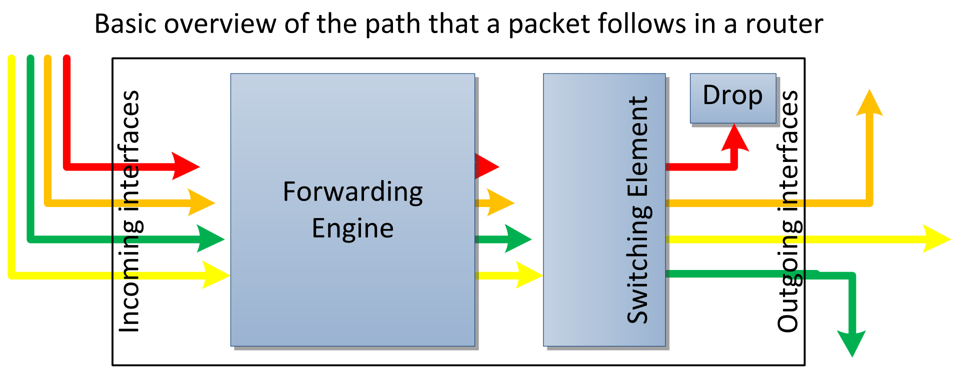


Figure 2.1: *The router receives packets from its incoming links. For each packet a lookup is performed on its destination address. Depending on the selected action with longest prefix the packet is forwarded to one of the outgoing interfaces or the packet will dropped, or what the action may specify.*

The forwarding engine, or routing table, stores address ranges denoted by there prefix. Where a prefix of 1011xxxx covers a range of 10110000 to (and including) 10111111. For each entry a possible output port is stored. A lookup in the routing table determines the output port where a packet with a specific destination address should forwarded to. A packet might encounter numerous routers before the final destination is reached.

The forwarding engine has to find the most convenient outgoing link for a received packet, which is the most suitable path. Whenever a direct link is not found then the packet will be send to a route which has the largest matching prefix. The routing algorithm has some lookup functionality which is able to find the route with the longest prefix matching the address. With each prefix an action is defined, this *action* determines to which output port the respective packet should be sent to or, what other action the router should perform.

The IP Lookup algorithm needs to store all known prefixes in such way that a lookup of an address is done as efficiently as possible. It is possible to store the prefixes as a list and for each lookup check every element in the list and select the longest matching prefix. But with an increase in number of prefixes, also the time for each lookup will increase with the same pace. There are other methods to store the prefixes in a routing table, for example a binary tree, see Figure 2.2.

When the prefixes are stored as a binary tree, then a lookup would correspond with the traversal of the tree. At each level a extra bit of the prefix is compared to the destination address, after which it can continue to the correct child. The leaf nodes will then correspond to a specific prefix. The advantage of such a structure is that the time required for a lookup will not increase with the same pace with the number of prefixes in the routing table.

The tree structure also gives much possibilities for further optimizations. In the next section two methods, the Radix Trie [16] and the LC-Trie [15] are discussed. Both of them use the binary tree as structure and apply optimizations to make the tree smaller. Other structures are also possible, as will be discussed with the Range Tree [16] and the Range Trie [19] in Section 2.2.3 and 2.3.

2.2 IP Lookup mechanisms

When the prefixes of a routing table are stored in a tree structure there are many optimizations possible. Therefore some implementations of tree structures which are used as routing table are pointed out. These other mechanisms are specifically chosen as the results of the optimizations are clearly visible and very well comparable to the optimizations done by the Range Trie. The two methods, the Radix Tree and the LC-Trie, are possible implementations of routing tables within the Linux Kernel [2].

During the explanation of the IP Lookup algorithms, the optimizations are performed on the same base-tree (Figure 2.2) which is a normal binary tree. By using the same tree to perform the optimizations to, the result is a clear visual indication what the different methods actually do to a given tree. The methods are ordered by complexity, starting with the Radix Tree and then the LC-Trie. After these trie-based structures the Range Tree [24] is discussed. The Range Tree uses a different method to traverse the tree during a lookup. Finally the Range Trie, which is an optimized version of the Range Tree, is discussed in Section 2.3.

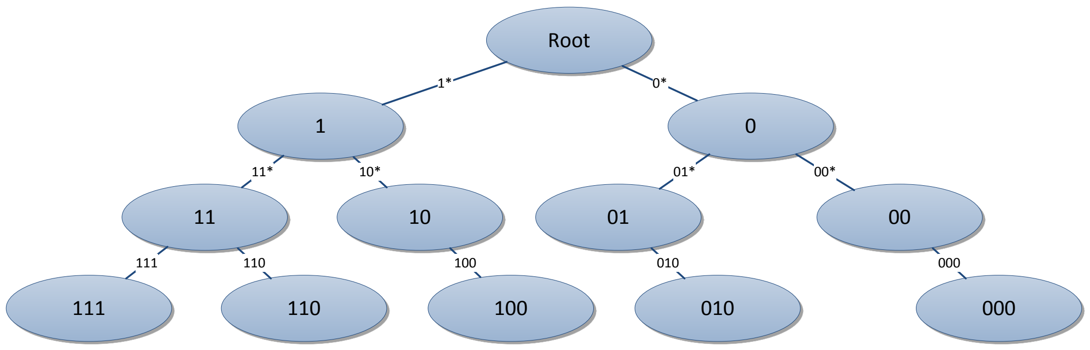


Figure 2.2: *A simple binary tree on which the optimizations for the different methods are performed*

2.2.1 Radix Tree

Radix Trees are used by default on most Linux distributions [2] and are originally called Patricia trees and are also called path-compressed tries. When starting with a random binary tree it is possible to have intermediate nodes which have only one child. These nodes are not really necessary because there is only one child the lookup can go to, therefore it is possible to remove them, or merge them with their child, see Figure 2.3. By merging the intermediate nodes with their single child sparse populated areas in the tree will be compressed very well [16], which in turn leads to a reduction in memory size.

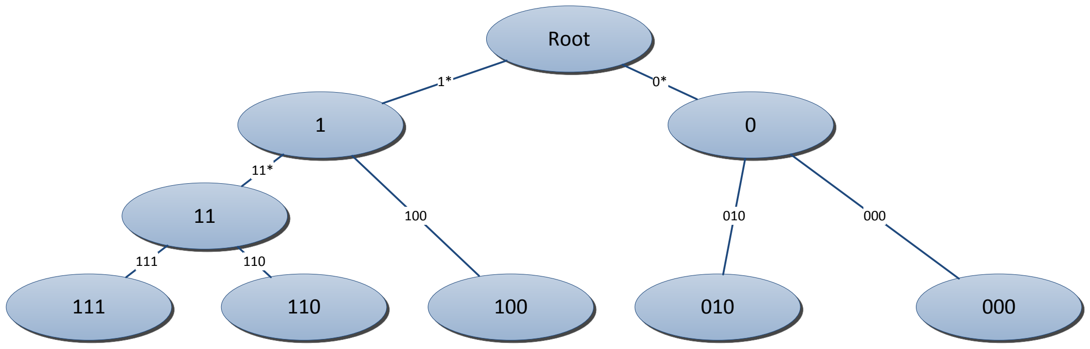


Figure 2.3: *To get the Radix Tree, the paths of the original binary tree are compressed, reducing the memory required for storing the tree*

2.2.2 LC-Trie

The LC Trie (or Level Compression Trie) uses the same method as the Radix Tree to compress the path of the base tree, its difference lies in the compression of levels [15]. Whenever all children of a node are fully connected, so both children of a node have two children themselves, then the node and the children will be replaced by a new node which has all children of the children. So instead of requiring 2 levels with both i branches per node, the replacing node will have 2^i children, as displayed in Figure 2.4.

The LC Trie has a higher complexity for a comparison at a node, as more than the default number of children are possible. Due to this increased complexity it can decrease the number of levels of the tree and with that the number of memory accesses for each lookup. This level compression works recursively, in such that a node with 2^i children which all have 2^i children will be replace by a new node with 2^{i+1} children. Although the comparison in each node is more complex the latency of a complete lookup can decrease due to less levels.

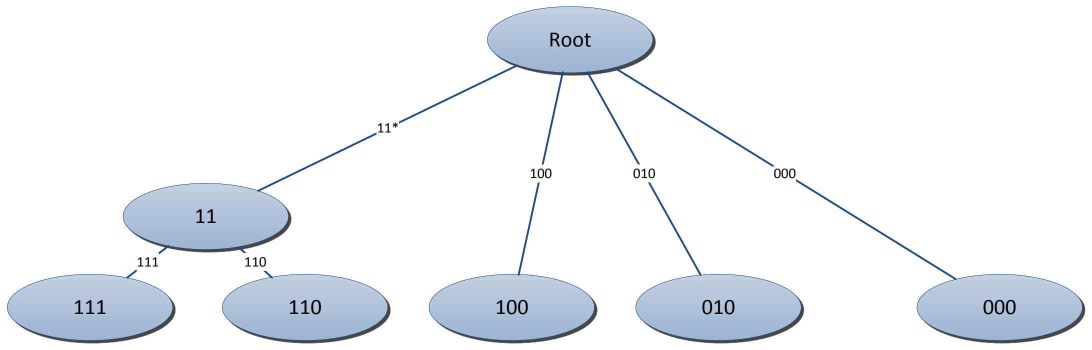


Figure 2.4: *In comparison with the Radix Tree, the levels are compressed to generate the LC Trie. Compared to the base tree, both the paths and the levels are compressed giving both a memory and latency optimization.*

2.2.3 Range Tree

The previous two tree implementations require a match of one or possibly more bits in every node. With the Range Tree this is not the case. The Range Tree [24] does not check on actual bits of the address but matches the complete address in each node completely and continues to one of the children depending on the value of the address [24]. In Figure 2.5 an example of a Range Tree is depicted. A downside of this method is that complete addresses have to be compared at each node. A positive side is that the Range Tree very much resembles the ranges of IP addresses in the IP address space. As opposed to the prefixes in the previous Trie structures.

A Range Tree can have multiple bounds in each node. A lookup will mean a traversal of the tree and at each visiting node the given address is compared to the bounds of the nodes. Take Figure 2.5 as an example with only one bound per node. When the address

01011 is looked up it will be compared to 01100. As 01011 is smaller than 01100, the lookup continues to the node on the left. When compared to 01000 the next node will be on the right with 01010 to 01011 as selected range.

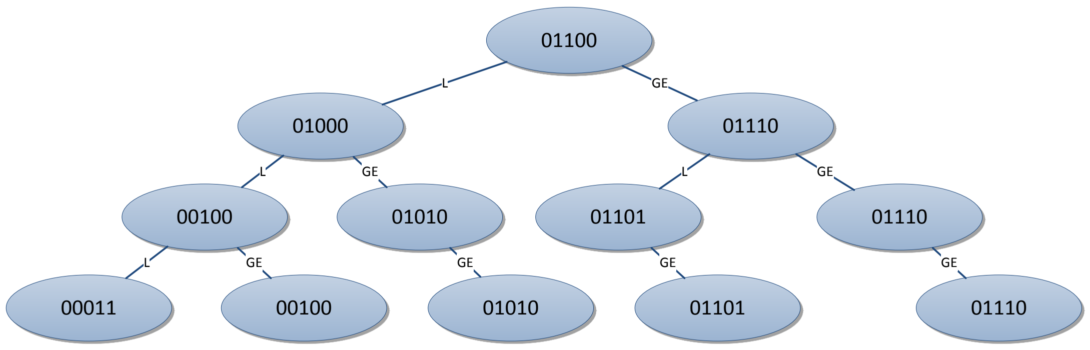


Figure 2.5: *The Range Tree does not compare one or more bits to find an exact match. The complete address is compared to the bound in a node. The answer, Less or Greater/Equal determines the next node to visit.*

Now the other IP Lookup algorithms are explained it is clear that the final tree becomes less deep when a node can have more than 2 possible children. Also, with multiple optimizations the total complexity of a possible design becomes higher while the lookup performance will be optimized with quite a good degree. The ideas proposed by the Range Tree seems to be more usable for IP lookup as, ranges with specific prefixes are the final goal. The Range Trie combines the range lookup with optimizations to reduce the number of bits to be compared at each node [20], as will be discussed in the next Section.

2.3 Range Trie

The original method for using a Range Trie [6] is developed at TU Delft, as well as its initial designs and the additions to support Longest Prefix Match [22] and [12]. TU Delft is also holder of the patent of the general method, system, and its data structure [18] as obtained in October 2010.

With the Range Trie a fully scalable method and implementation is created to replace the current used Longest Prefix Match algorithms in internet and network routers. The biggest advantage of this method is that the depth of the tree is decreased substantially. This means that every lookup, or other operation, require less memory accesses. The lookup latency and power consumption will be decreased consequently.

First the default structure of the Range Trie will be explained, including how lookups and updates are performed. After that one of the biggest disadvantages of this method is explained and a solution is proposed. Finally both parts will be used together in the design and implementation of the Range Trie route lookup mechanism.

2.3.1 Basic Range Trie structure

The Range Trie uses ranges, just like the Range Tree, to traverse the tree. In the Range Trie however partial, rather than complete addresses are compared in each node, the algorithm that builds the tree attempts to eliminate as many bits as possible to reduce the number of comparisons needed. While building the tree the heuristic proposed by [6] starts with all known prefixes and starts building the Range Trie from the bottom up storing as many prefixes as possible in each level. Each node in the layer above the last can have up to 29 children and these intermediate bounds are decided upon by the algorithm. The algorithm selects internal bounds to minimize the number of bits needed per comparison, and consequently maximizes the number of bounds per node. To find the *ideal* intermediate bounds the algorithm will take several optimizations into account, as presented in [19] and [20], these optimizations are:

1. Omit common node prefixes
2. Shared common prefixes
3. Shared common suffixes
4. Omit zero suffix
5. Address alignment

By including these rules the number of bit-comparisons can be significantly decreased for each bound. In the next subsections the reasoning behind, and application of these five rules are explained. The proof for each rule can be found in [18]. For the coming explanations the following notation is used:

- The node currently looked at is denoted by N and has address A_N with neighboring node N+1 with address A_{N+1}
- The node N has x internal bounds, with addresses A_x
- The incoming address is denoted by A_i

2.3.1.1 Rule 1: Omit common prefixes

When in node N, then A_i is at least as large as A_N and all A_x are equal or larger than A_N. If A_N and A_{N+1} have some common prefix P, then it is for certain that each bound A_x and address A_i also have this common prefix. Therefore it is not necessary to check for this common prefix part and it can be omitted during the comparisons, as can be seen in Figure 2.6.

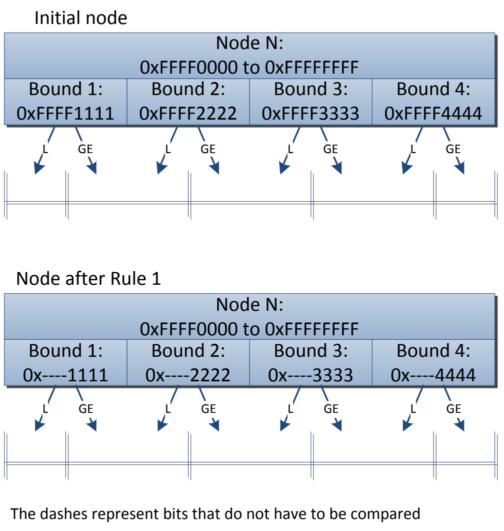


Figure 2.6: *Visualization of Rule 1: upper part is the original comparison and at the bottom the comparison has been converted using Rule 1.*

2.3.1.2 Rule 2: Shared common prefixes

Within a node N all bounds can have an equal, shared, prefix. As with Rule 1, except for the prefix requirement at A_N and A_{N+1}. In such case it is not necessary to compare the shared prefix for every bound comparison individually. Instead the shared prefix is compared separately and only when the prefix matches then the rest of the addresses have to be compared as shown in Figure 2.7. When the shared prefix matches the most significant bits of A_i then the other bounds with the equal shared prefix will have to be matched, but only the bits not already matched with the shared prefix.

2.3.1.3 Rule 3: Shared common suffixes

When the bounds A_X share a certain suffix, then this shared part can be compared only once and used in the different individual comparisons. From Figure 2.8 the resemblance to Rule 2 is clear, but the interpretation and location of the shared comparison is of course different then during Rule 2 as this is a suffix instead of a prefix. The shared suffix only needs to be compared when the other bits of a bound match exactly, if this is not the case the this comparison is not necessary as the final result is already known.

2.3.1.4 Rule 4: Omit zero suffix

When it is known that a bound ends with a certain amount of zeros, there is no need to compare them, as the bounds are to and not including. When an incoming address matches the bound-address without the zero suffix then it already is certain that it will

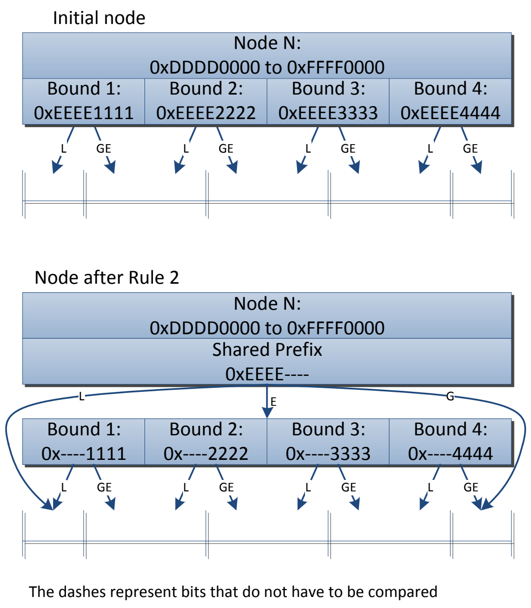


Figure 2.7: *Visualization of Rule 2: upper part is the original comparison and at the bottom the comparison has been converted using Rule 2.*

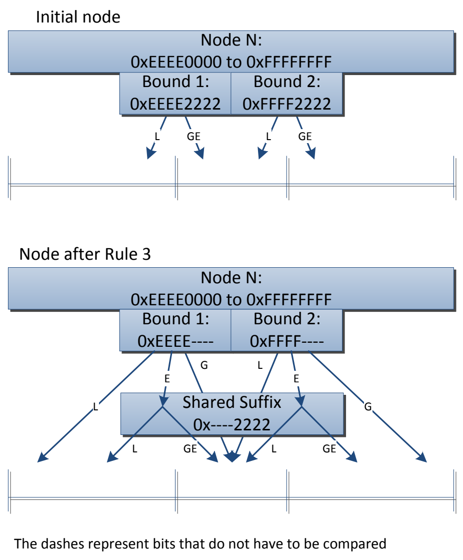


Figure 2.8: *Visualization of Rule 3: upper part is the original comparison and at the bottom the comparison has been converted using Rule 3.*

map to the Larger or Equal part of that bound. In Figure 2.9 the following example is displayed.

When there is an A_i equals 0xAAAAF000 then and in a certain node there is a bound A_X with address 0xAAAA0000. If there is a complete comparison then A_i will go to the Less side of A_X. When this rule is applied, the comparison will be done with A_i=0xAAAA and A_X=0xAAAA which will also map to the Less side of A_X.

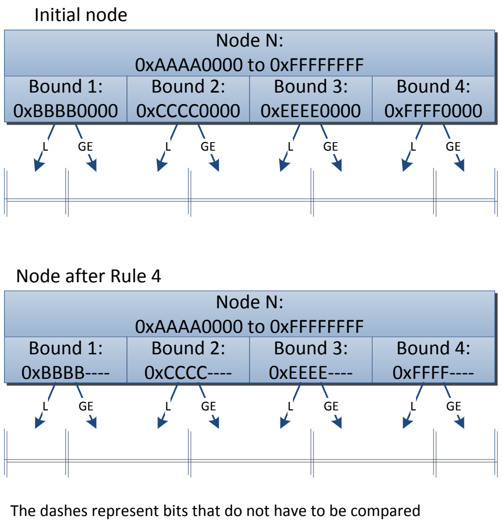


Figure 2.9: *Visualization of Rule 4: upper part is the original comparison and at the bottom the comparison has been converted using Rule 4.*

2.3.1.5 Rule 5: Address alignment

An incoming address A_i should map to a certain address range. Instead of comparing the remaining bits of A to the two bounds of the range it is also possible to subtract the left bound of the range from all bounds in the certain nodes and the incoming address. If the subtracted values are compared, it will give equal result as to a comparison with the original addresses, but it might be that the subtracted value has lesser bits to compare. A good example is given in Figure 2.10 where 32 bit comparisons are converted to 8 bit comparisons due to this rule.

While subtracting the value from the incoming address it can be said that the width of the comparisons will be $L = log_2(A_N - A_{N+1})$ at most. Therefore it will not be necessary to subtract more bits of the incoming address than needed in the comparison, as more bits will just not be used. This is why the incoming address in Figure 2.10 is subtracted with 0xF0 while the bounds are subtracted with 0x3FFFFFF0.

Rules 1 to 4 can be used independently from each other, for rule 5 there are some issues

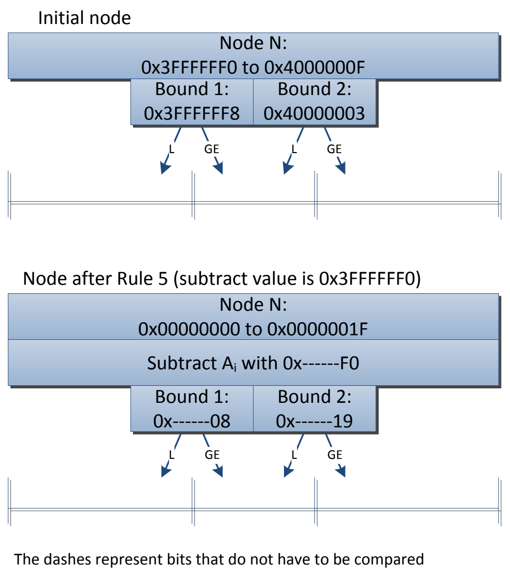


Figure 2.10: *Visualization of Rule 5: upper part is the original comparison and at the bottom the comparison has been converted using Rule 5.*

to keep in mind. In such that rule 5 can be combined with rule 1 to maximize the common node prefix, but rule 5 should be applied before rule 2, such that no shared prefix is detected while the subtraction will probably change this value. Rule 3 and 4 can be applied independently to rule 5. In the example of the Range Tree in Figure 2.5 only one bound per node was taken into account. If this example is used for the optimizations of the Range Trie the tree will look like Figure 2.11. It will further depend on the implementation if the nodes without comparisons actually exist.

2.3.2 Performing a Lookup

The main function of the Range Trie hardware design is to perform lookups. During a lookup command the Range Trie will receive an address which is to be looked up in the tree. To do so, the design traverses the tree, using the destination address of the incoming packet, and with each visiting node it compares the required bits from the node and the asked address. At each node the Range Trie can have 29 siblings, and the design should be able to determine which of the 29 possible children will be next node to visit. A possible traversal down the tree is given in Figure 2.12. The result of a lookup will be an action. An action tells the switching element in the router what to do with the packet, either send it to a certain interface, drop the packet, or some other action.

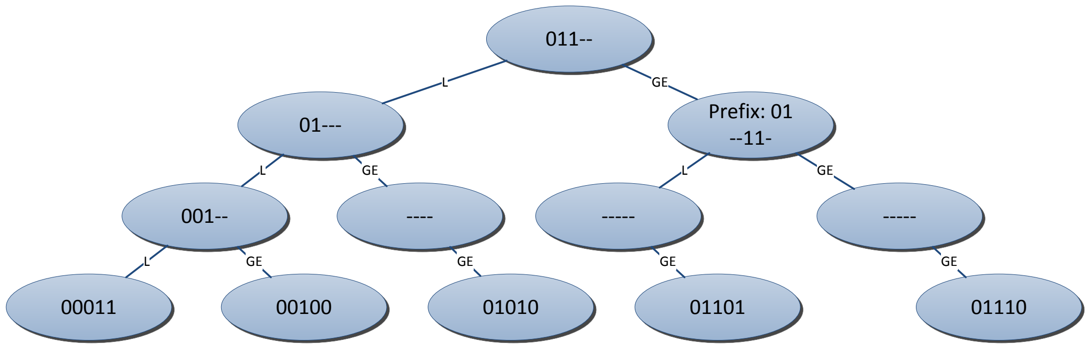


Figure 2.11: *The Range Trie with Figure 2.5 as basis with all possible rules applied. Depending on the implementation also the empty nodes can be omitted. The Range Trie heuristic will give different results as multiple bounds are supported and nodes are generated bottom-up and not performing the optimizations on an already existing tree.*

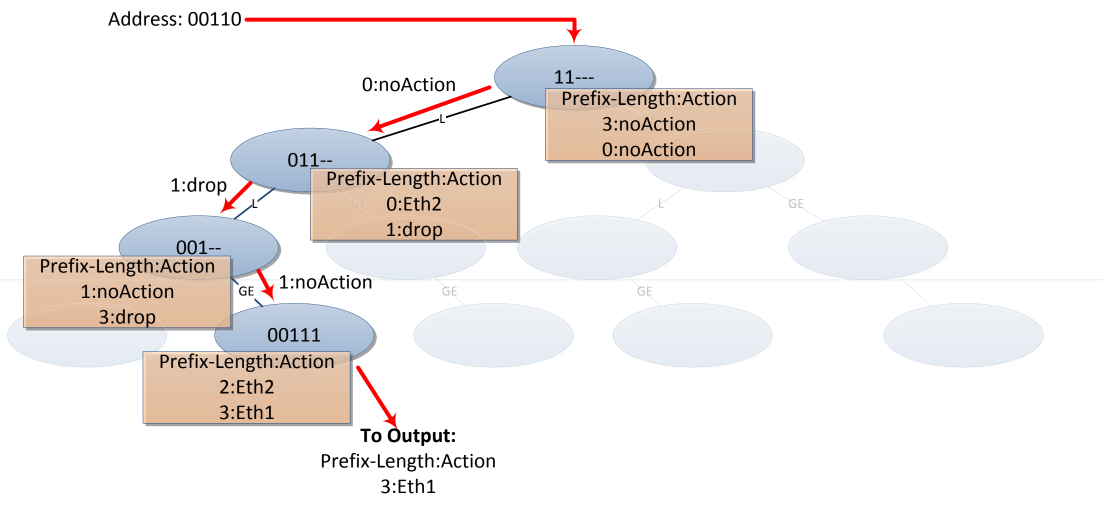


Figure 2.12: *Example of a route lookup including the LPM-support, at each node the prefix-length is checked. Whenever a larger or equal value is found the value is updated and the new value with its action will be forwarded.*

2.3.3 Longest Prefix Match

If a destination address maps to multiple ranges, the lookup will have to select the ranges with the largest prefix equal to the destination address. In Figure 2.12 it can be seen that a prefix length can be stored at every level in the tree and as soon as a larger prefix is found, then this value is taken down with the lookup. With the prefix lengths, also an action pointer is stored with each bound. With this action pointer it is possible to select the action corresponding to the selected prefix. If a visited node does not have a larger prefix-length for the given address then the action and prefix-length from the previous levels is passed on down to the next node. During a lookup the final prefix length and action are reported.

2.3.4 Performing an Update

Having a Range Trie design in hardware which can do fast lookups is already promising, but it has to support updates for actual use. To perform an update two addresses are required which are the bounds of the range of addresses which are possibly updated. Possibly, because it can happen that for certain addresses within the updated range a larger prefix than the prefix of the update is already known. In that case the update is not performed for those addresses.

In order to know if the addresses within the range need to be updated they first must be checked, therefore a lookup on both the low-bound and the high-bound of the update is performed. Together with the information from the lookups, it is possible to tell which elements in the ranges require updating. As the bounds may split in the tree, which means that they do not follow the exact same path, also the updates are split and performed in two cycles. Because two lookups and two updates are required for a normal update it is key that there are no other commands executed right after the two update commands as they might interfere with the two update attempts.

With the updates, the information at the existing nodes can be changed so there is a way to update prefix lengths and actions. It is possible that new links are discovered, when this happens a new prefix should be added to the routing table. As many of the optimizations require knowledge about the complete node with multiple bounds it is not very straight forward to add a new prefix to the structure. The tree might require a complete rebuild for each update. As this will take too long a different method is proposed, the spare levels. The fixed part will follow the basic Range Trie structure as explained in this Section, but does not allow the on-the-fly insertion of new prefix bounds. The insertion of new prefix bounds is only possible in the spare part which allows a certain amount of levels to be inserted.

2.3.5 Spare levels

As it will be very complicated to include all optimizations as integrated in the fixed part of the Range Trie, the spare parts only includes rules 4 and 5. These rules are applied because no knowledge about neighboring bounds or the parent node is necessary and therefore no recalculation is necessary are a new bound is added to a node. The spare levels will contain small subtrees under the fixed part at the location where bounds are added, an example of this is shown in Figure 2.13. This allows for an optimization in the memory usage.

As the spare levels are only used when necessary, it would be a waste of memory if it uses the same memory layout as the fixed levels as many entries will remain empty. Therefore a memory management system is proposed for the complete spare part which allows memory blocks to be dynamically allocated to a certain spare level when required.

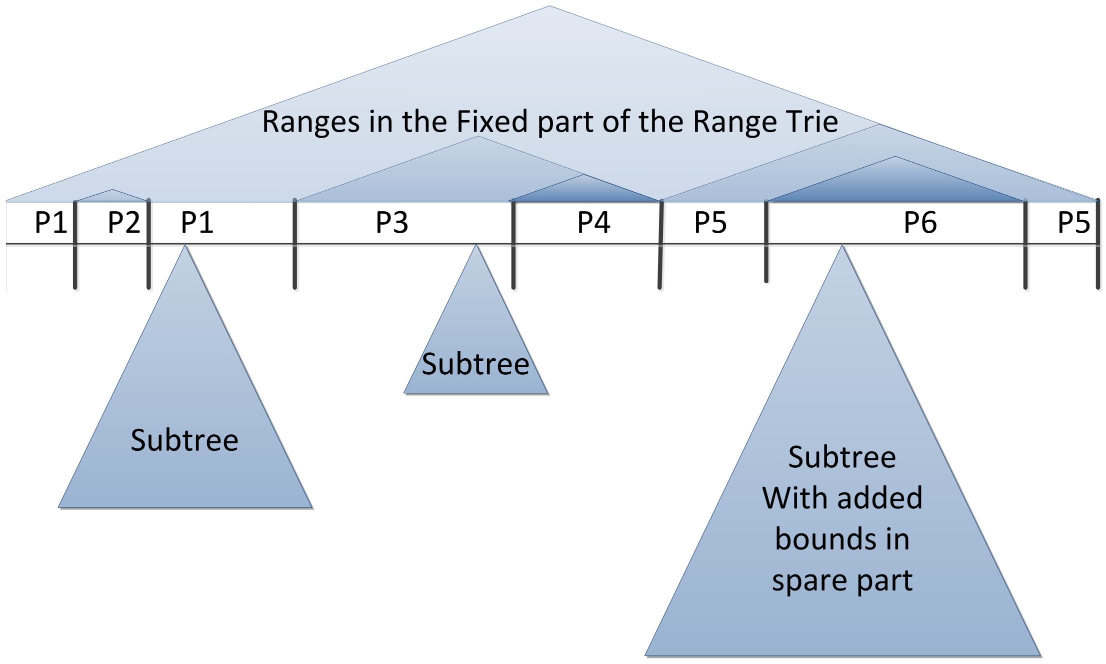

Ranges in the Fixed part of the Range Trie
P1
P2
P1
P3
P4
P5
P6
P5
Subtree
Subtree
Subtree With added bounds in spare part

Figure 2.13: *When bounds are added to the Range Trie, these will be stored in the spare part as individual subtrees under the leaves of the fixed part. In the leave node of the fixed part, a memory lookup is performed to retrieve a possible pointer to the subtree. If this pointer exists, then the subtree will be traversed. The subtree can contain multiple levels and one single leave node in the fixed part can contain multiple subtrees.*

With this method it is possible to have either many small subtrees (with few levels) or lesser large subtrees (requiring many levels).

This different memory management requires an additional lookup for each spare level. This additional lookup is from a maintained next-pointer memory. Which takes the next-pointer from the previous level and from that lookup the base-pointers in the dynamically allocated memory for the next level. Why this extra lookup is necessary is explained more elaborately during the implementation in Chapter 3.3.5.

The structure for the spare levels with just a few optimizations is chosen because it is easier to implement and it suffices. A large downside is that the tree can become unbalanced when many of the bound insertions point to the same side of the tree. When this happens the levels for a single subtree are filled relatively quickly without maximizing the memory usage. When a subtree is out of spare levels, no extra bounds can be inserted and the spare tree should be incorporated into the fixed part by rebuilding the tree. During this restructuring all bounds will have to be read and will be used as the input of the tree building function in software.

To maintain the availability of the hardware for lookups, the restructuring will be performed in software. Updates should be stalled during a tree rebuild, because updates performed in the hardware while restructuring will not be visible when the new restructured version is being implemented. Therefore updates should be stalled until the restructuring is finished. Performing the restructuring in hardware would take many cycles with a high complexity, while there currently exists an efficient algorithm to create the tree in software. Next to the availability of lookups during the restructuring, this is another reason why the rebuilding of the tree is currently executed in software.

2.3.6 Performing a Lookup in Spare levels

In general, the lookups in the spare levels are performed in the same manner as in the fixed part of the tree. There is just a slight difference on how the address of the next node is calculated, this is due to the dynamic allocated memories and more attention is spend on this in Section 3.3.2.2. In the spare levels the number of comparators in each level to compare the incoming address to the stored compare values is equal to the number of comparators in the fixed levels. As there are less optimizations on the compared values, the number of bits compared for each child is higher. This has as consequence that in the spare levels each node can have up to 9 children, which is smaller than the number of possible children in the fixed level which can have up to 29 children.

2.3.7 Performing an Update in Spare levels

Next to the lookups the updates are also more or less the same as in the fixed part, except that it is possible to add and delete bounds. During the initial lookup cycles the spare part checks whether the update will cause a new bound to be added. If this is the case then the spare part will allocate memory for this bound. The bound itself is added during the following update cycles.

With the allocation of the memory for a new bound the design will check if the bound can be placed in the current node. Whenever the node is already full the bound has to be added to the next node. If this next node does not yet exist, it will be created. The creation of a new node can require allocation of a new memory block to that spare level.

Next to bound additions it is also possible that an update will cause a bound to become unused. In the fixed part this bound cannot be deleted physically as the rest of the structure can rely on it. In the spare part bounds are added independently and can thus be deleted independently as well. Even more, when a complete allocated memory block is unused it can be recycled in order to use it for another subtree when requested. During the creation of the hardware implementation a severe bug was found in this reasoning as the pipeline might need an extra, fifth, cycle for updating the base pointer tables in the spare part. More about this can be found in Section 3.3.6.

2.4 HTX Platform

The platform on which the Range Trie design will be implemented is a general processing machine which has support for hardware acceleration of functions thanks to the work in [3]. The implementation of the hardware and corresponding driver lets normal program functions to be executed in hardware. It gives the software programmer the ability to run a function in hardware as if it is a normal C function, including the passing of parameters. The most common way to use these parameters are pointers to main memory as it is not yet possible to let the function return a specific value. The use of pointers gives the opportunity to let the accelerated function write back the result to main memory. In this functionality it is very much like the Molen designs [23] but with the Molens embedded PowerPC-processors are used which are part of the FPGA instead of a general CPU in a relative computing environment with an HyperTransport Bus [4], also referred to as HTX bus.

The FPGA used is a Virtex-4 HTX board [9] which has an interface to the HyperTransport Bus [17]. The resulting platform is able to directly access the main memory which is shared with the host using virtual addresses. The core of the platform translates this virtual address to the real memory address before it is send over the HyperTransport Bus. This makes the memory access from the accelerated function as implicit as possible as no explicit address translation or prefetching has to occur.

Whenever the platform attempts to access main memory, it only has access to the pages belonging to the application calling the function, this prevents the accelerated function from writing and corrupting memory belonging to other programs or the operating system. The driver also takes care of swapping the correct page to accessible memory when necessary. In order to have the memory that is being accessed by the hardware function available and it does not have to wait for the memory swap during runtime. The HTX hardware uses interrupts to inform the driver of TLB misses and to tell the originating application that the function has finished. This means that other applications are not blocked by the hardware function as the driver does not need to check continuously whether the hardware function has already finished.

There are different implementations of the HTX platform which will be discussed in Section 2.4.1. The chosen implementation has influence on how accelerated function gets the data from main memory and choosing the right version might speed up the final execution. After that the interfacing between the hardware core of the HTX platform and the accelerated function is discussed in Section 2.4.2. Finally in Section 2.4.3 the usage of the hardware accelerated function from software will be explained.

2.4.1 Implementations of the HTX platform

Currently there are three different implementations of the HTX platform. The main differences between the three options is the way how main memory is accessed and how the data is retrieved. In the next sections the three different implementations will be revealed and depending on the differences and the demands of the function the best version for the Range Trie implementation will be chosen. The first implementation is the generic implementation and the other two are optimized versions of the first one.

2.4.1.1 Generic HTX platform implementation

The simplest implementation of the hardware designs for the HTX platform is displayed in Figure 2.14. In the figure the different parts of the structure are depicted. These will be explained in the next few paragraphs. It is the simplest design as no optimizations are made with respect to memory access.

Starting with the communicating elements, IO, DMA Read and DMA Write. The DMA Read and DMA Write take care of all required data communication to the CCU, the accelerated function. In this case 64 bits elements can be requested from or written to main memory. The IO unit is responsible for all host initiated communication and for transferring the values of the exchange registers.

It is possible for the software to send a memory address via the exchange registers to the CCU as function parameters. In order to get the real memory address from the virtual address a Translation Look-aside Buffer, TLB, is maintained in hardware. Every time a new virtual address is given for reads or writes the Address Translation unit will try to translate the table using the TLB. Whenever a TLB miss occurs, an interrupt is raised by the Interrupt Handler to find the correct translation from software, which will then be stored in the TLB for future availability. Interrupts are also raised whenever the accelerated function has finished its operation.

This generic implementation is able to request only one data element of 64 bits per request. With the other implementations multiple elements near the same memory address are requested to minimize the communication overhead. When functions request data, most of the time this data will be stored near each other, for example in a row if an array is requested. With these optimizations this locality will be exploited.

2.4.1.2 HTX platform implementation with Caches

A read requires a read request to the software and there is a delay until the data is received. It might be efficient to reduce the number of individual reads, to have the delay only once. Instead of requesting 64 bit elements one by one, a full cache line of 512 bits will be requested, these data elements will be stored in a local hardware cache as seen in Figure 2.15. When other data packets are requested by the CCU, first the hardware cache is checked. Whenever there is a hit in the local cache a request to the host is not necessary. Of course there will be a check when data is written, in such that the local cache might have to be updated as well.

2.4.1.3 HTX platform implementation with Queues

Using caches to read cache lines instead of single elements is quite efficient, but still the full bandwidth of the HyperTransport Bus is not used. In Figure 2.16 the block diagram

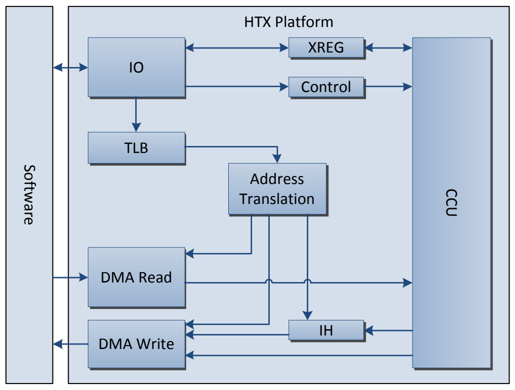


Figure 2.14: *The simplest implementation of the designs for the HTX platform, with direct memory access without any optimization with respect to memory access.*

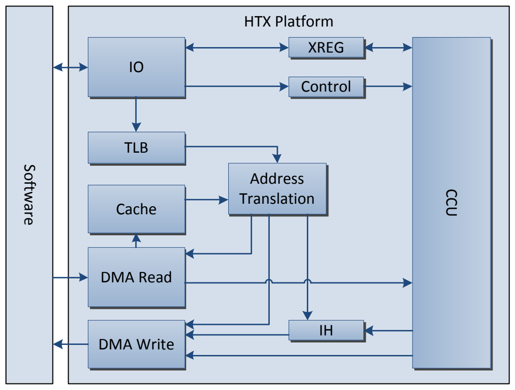


Figure 2.15: *In order to save time waiting on data elements from the host, a cache can be used in combination with a larger request. This works quite well if the next requested element maps to the cache line as requested previously.*

of the used and most complicated implementation is given. The most specific point with this implementation is that it uses queues for the communication between the host and hardware. This gives the hardware the opportunity to prefetch the data and store them temporarily in the queues.

The queued version is the most applicable implementation of the hardware platform for our purpose as the commands will probably be given in an almost streaming fashion. With the queued implementation the software has to store the commands in an array and when the accelerated function is started the array will be transported to the hardware very efficiently. After which the Range Trie can execute the commands as if it was streamed to the hardware.

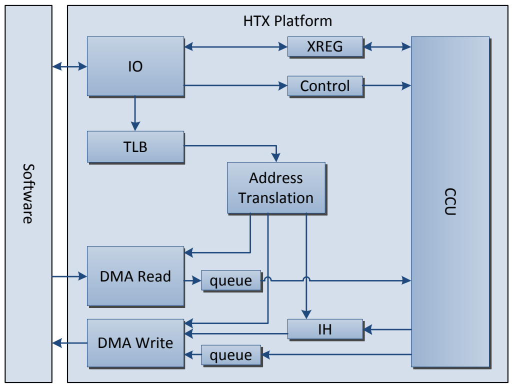


Figure 2.16: *With queues the HTX platform has the ability to prefetch data as early as possible. This is the most efficient way to get the data whenever many consecutive elements will be requested, for example an array.*

2.4.2 Accelerator Interface

Within the hardware implementation there needs to be communication between the hardware related to the HTX platform, the core, and the function that is being accelerated. In Table 2.1 the interface signals are listed and described. Most of the signals are used to transfer information from the software to the function and vice versa. Also some signals are used for the exchange registers which can also be used to communicate with the software. The top four signals are the control signals, they tell the function when it should start and also the stop or done signal from the hardware to the software is included. The interface resembles a lot to the interface that the Molens use, this is done so that it is easier to use (almost) the same function on both platforms.

2.4.3 Interface to software

From the software side there is a very programmer friendly way to execute an accelerated function from within the FPGA using this method. Here a short list of steps is included to successfully use the FPGA functionalities from an executable created in the programming language c.

- include a header-file `#INCLUDE molen-htx.h`
- put the replace attribute above the function which has to be replace by the hardware function: `__attribute__((user("replace")))`
- while compiling the c-file make sure the gcc compiler is used with with the special module. This module takes care of the actual replacement of the dedicated function to the correct hardware calls.

Table 2.1: *Interface Signals between CCU and HTX core*

Signals	Width	Description
start	1	The start-signal for the accelerated function. This means that the parameters are available.
done	1	When the accelerated function has finished this signal will inform the software so it can continue.
MIR	64	Control signals from Molen, not used in the HTX platform
status	2	Control signals from Molen, not used in the HTX platform
xreg addr	10	Shared read/write address for the exchange registers
xreg read data	64	Read signal for the given exchange regsiter
xreg write data	64	Write value for the given exchange register
xreg write enable	1	Write enable for the exchange registers
mem addr	64	Virtual memory address where to read/write data to/from
mem size	64	How many elements will be read/written starting from the address
mem read data	64	Read signal for the memory
mem read enable	1	Issues a prefetching of the set amount of data-elements from the given memory address
mem read next	1	When a value is read, this signal will ask the core for the next to read value
mem read ready	1	A read-enable may only be given when the memory-controlleris ready for such an operation
mem read valid	1	This indicates if the value on the read-data signal is valid
mem write data	64	Write values for the memory
mem write enable	1	Sets the write buffer to write the given amount of data elements to the given memory address
mem write next	1	Write the next data element to the send queue
mem write ready	1	A write-enable may only be given when the memory-controller is ready for such an operation
mem write full	1	When a send queue is full, the accelerated function should stall any writes untill this signal is empty again

Because this system offers almost seamless implementation in any c written program it is very interesting to accelerate kernel functions such as the routing tables for routers. As most users tend to switch programs quite much on-the-fly reconfiguration of the accelerated function is required when normal programs have the ability to utilize hardware acceleration. This is one of the subjects that is currently investigated, for further development of this system.

2.5 Summary

Much research has already been done on IP Lookup data structures for network routing. Some of the more known structures and algorithms were discussed in Section 2.2 such as the Radix Tree which is used as default Linux routing table implementation. Compared to these methods the Range Trie has more optimizations which should lead to a faster route lookup. As previous work has not developed a working prototype, no measurements of a real Range Trie system have yet been performed.

In this thesis, the Range Trie design will be implemented in a reconfigurable platform. The HTX platform has been discussed in this chapter mentioning most the important details of the system. One of the discussed elements are the three different implementations. The HTX platform which uses queues to prefetch large amounts of data seems to be the most efficient implementation to use in conjunction with the Range Trie design. As the Range Trie is a streaming application which can handle a new command almost every cycle.

3 Implementation

The fully functional implementation of the Range Trie on the HTX platform consists out of three parts; (1) support of the HTX platform, (2) communication to the Range Trie and (3) a fully functional design of the Range Trie. In Chapter 2 the background information about routing tables and different route lookup methods including the Range Trie were given. It is clear why the Range Trie has great potential to solve the problem of current route lookup mechanism that do not scale well with the currently increasing demands due to the increasing number of entries and the coming increase in address sizes. Next to the algorithm which is implemented, also the HTX platform has been explained with its support for reconfigurable functions which can be called from software by using system calls.

In order to have a complete implementation our attention is required at the hardware platform because the Range Trie designs currently operate on a 50MHz clock which was not supported by the HTX platform. The way this problem is handled is discussed in Section 3.1. In the hardware design, communication between the hardware platform and the Range Trie design is a very important issue and this will be discussed in Section 3.2.

In order for the Range Trie designs to be implemented, the HTX platform is modified and a wrapper is implemented first. During the implementation the following subjects will be discussed in Section 3.3:

- **Memory structure**: In the initial Range Trie designs LUT memories are used. Because the LUT memories are implemented in logic blocks this uses many resources. Therefore BRAM memories, which use specific memory elements, will be implemented. The change in memory type has issues in the pipeline of the Range Trie as LUT memories have a faster response than BRAM memories. The issues are discussed in Section 3.3.1.
- **Memory Utilization**: In the Range Trie designs there are many different memory blocks. In Section 3.3.2 the usage of the different memory blocks and their contents is explained. As not all functionalities in the Range Trie design were working correctly the memory utilization was checked for consistency such that the lookups and updates are performed correctly.
- **Actions**: The purpose of a lookup is that the switching element knows what to do with a certain packet. This action can be stored in different ways, for example in software or hardware. Both ways have their own advantages and the final implementation will be discussed in Section 3.3.3.

- **Spare levels**: In the original design, the spare levels were not functioning correctly. Some of the issues encountered and discussed in Section 3.3.4 are:
  - **Out of order bound additions**: During run-time, updates will come randomly and bounds may be added in any order. In the initial design only in order bound additions are supported and the spare levels will be used quite inefficiently. Therefore out of order bound additions are now supported in the final implementation as described in Section 3.3.4.1.
  - **Addition of multiple bounds per update**: When a new prefix with large prefix-length is inserted into the spare part it might be possible that both bounds do not yet exist. The earlier design does not support double bound additions inside a node, this is discussed and implemented with Section 3.3.4.1.
  - **Bound deletions**: When a prefix is deleted is can happen that a bound will be unused. To increase the memory efficiency this bound should be removed, in order to allow a new bound to be added in that location. Section 3.3.4.2 discusses this added functionality.
- **Adding stages in the pipeline**: Due to the implementation of BRAM memories an extra stage in the pipeline is necessary for the spare levels as discussed in Section 3.3.5.
- **Extra cycles required for updates**: The original design did not allow bound deletion. During the implementation of bound deletion, also recycling of the memory blocks in the spare part was implemented. This recycling of memory blocks requires an extra cycle in the update bubble as will be discussed in Section 3.3.6.
- **Retrieval of the tree**: Whenever the spare part is full or unbalanced, the tree might require a rebuild. To rebuild the tree in software the complete tree information of both the fixed and spare part should be known to the software. Therefore extra commands are implemented and the software tree information is available for the software as described in 3.3.7.
- **Interface to software**: With the HTX platform, there is a direct link between the software and hardware. The software will pass any received lookup request, update or other command to the hardware and will react on the results as necessary. For the communication between the hardware and software a communication scheme is designed and discussed in Section 3.3.8. This section also includes how the software should initially program the Range Trie before it can be used as forwarding engine.
- **Pipeline of the Range Trie design**: During the implementation of the Range Trie, the original pipeline is modified. Section 3.3.9 shows the new pipeline including all modifications.

3.1 Hardware platform

The HTX platform as proposed by [3] consists out of two different parts, the core and the accelerated function. The accelerated which has to be accelerated, in this case the Range

Trie design, is also called the Custom Configured Unit or CCU. The core of the design will manage the communication to the host, either direct via memory or via the exchange registers, but also hosts the clock management for the CCU. The CCU, the Range Trie design, will communicate with the software through the core of the HTX platform and the HTX platform will provide the Range Trie design with all required signals, including the clock signals, the commands from software and the interfacing with main memory. Normally the HTX platform expects its CCU to run with a 100MHz clock.

The Range Trie design is currently running at a 50MHz clock which is not the default CCU clock for the HTX platform. The core uses multiple clock frequencies internally for different components, in such that the core runs on a 100MHz clock at the CCU side, but also 200 and 400MHz clocks are used for the communication parts and the control of the HyperTransport Bus (HTX-bus). In the core of the HTX platform there is a special module which manages all different clock signals, this had to be changed in order to retrieve a 50MHz clock for the Range Trie design.

As the current Range Trie design is built as a component where the lookup and update commands are fed into there will need to be some controller that operates between the Range Trie and the core of the HTX platform. The controller tells the HTX platform to get the commands from main memory and which feeds the commands to the Range Trie design. This controller is a wrapper around the CCU and will have to communicate with both the 50MHz Range Trie and the 100MHz core of the HTX platform. The wrapper will therefore run in a dual-clocked domain. In the next section the wrapper will be discussed more elaborately. In Figure 3.1 the total structure of the HTX platform is depicted including the wrapper around the CCU. This is the structure as will be used for the implementation of the Range Trie.

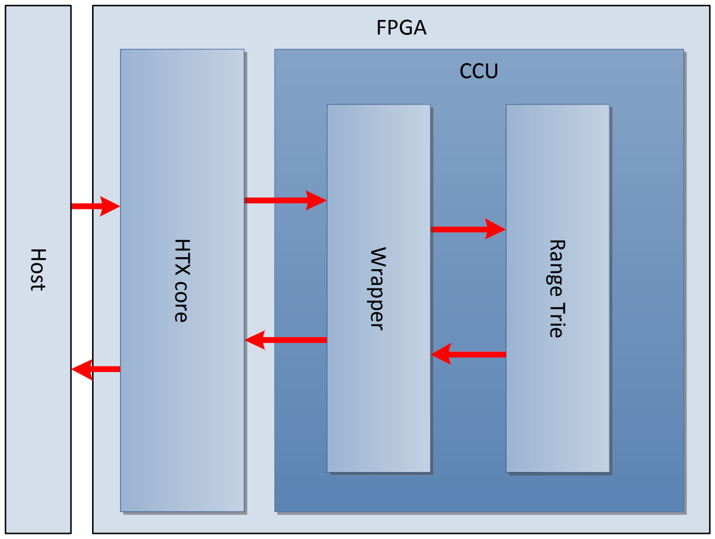


Figure 3.1: *Structure of the HTX platform including wrapper for the CCU*

3.2 Wrapper

In order to provide general communication between the host computer and the FPGA the methodology and implementation as in [3] is used. This provides general communication, but expects the CCU to be able to run at 100MHz. Currently this is not the case as the CCU is only able to run at 50MHz. There are different ways to implement an extra wrapper between the CCU and the core of the HTX platform, while ensuring a fast communication between the function and the host.

There are other reasons why a wrapper between the core and CCU is desirable. For example the communication bandwidth; the core can send and deliver 64 bit blocks every clock cycle while some commands require 319 bits of information which requires reordering and repacking of the received commands. Also some commands require more execution cycles or even a stall of the input for multiple cycles. The wrapper takes care of these requirements so the core and software can just send the information when available.

The wrapper basically takes all information of the core, corrects the format where needed and provides the information to the CCU respecting its requirements. It also handles the information from the CCU to the core in the same manner, it takes the result from the Range Trie and provides it to the HTX core in the correct format so it can be written to the main memory of the host.

In general there are three possible methods how the wrapper is setup; these will be explained and compared in the next three sections. After the different methods are investigated one of these will be implemented.

3.2.1 Wrapper 1: Dual Clocked Write/Read Queues in the HTX core

In the most ideal case the CCU will operate only on a 50MHz base as it removes the complexity of having one entity (the CCU) which has multiple clock domains. But the change is not as simple as only altering the HTX communication queues to permit multiple clocks, but also all other system signals need to be converted to 50MHz and vice-versa.

For example: the start-operation signal for the CCU is normally held high 1 clock-cycle (in 100MHz), so it is possible that the CCU (on 50MHz) will not see this signal. Next to the example of the start-operation signal there are many other signals which need attention to get this method to work, such as: end-operation, read-next, write-next, write-enable, read-enable, exchange-register-read, exchange-register-write and the list continues. Therefore this method seems simple, but there are quite some extra signals to take into account as described in Section 2.4.2.

3.2.2 Wrapper 2

To circumvent the added complexity by altering the core to communicate well with the 50MHz CCU, the wrapper will be adjusted to allow the reception and sending of all re-

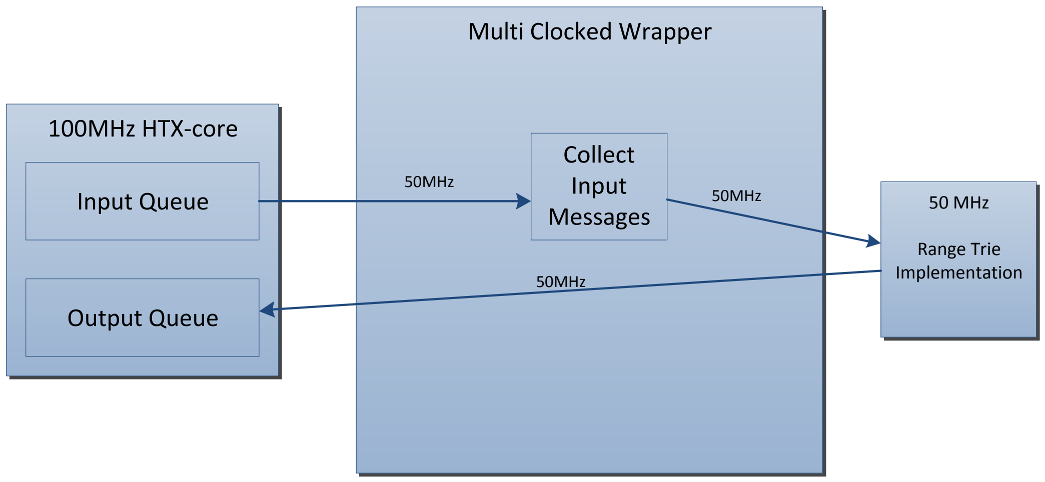


Figure 3.2: *Wrapper Implementation 1: Reordering in 50MHz domain. No extra buffers needed.*

quired signals in 100MHz to the core and in 50MHz to the CCU. To accomplish this type of behavior the wrapper will consist of multiple independent processes communicating with each other using multi clocked buffers. In this case the only information that needs to pass to the CCU are the commands (lookup, update, etc.) and the results of these commands need to be send to the core so they will be written to the host.

In Figure 3.3 the above implementation is given. As already stated the incoming commands might need to be reordered or repacked, therefore a specific block is included in the design. In this variant the repacking is done at 50MHz, which allows a buffer between the 50 and 100MHz domains with a width of 64 bits.

3.2.3 Wrapper 3

In the previous wrapper the reordering and repacking of the commands is done in the 50MHz domain. While there already is a 100MHz domain in the wrapper, why not perform these operations in 100MHz like in Figure 3.4? The problem with reordering at 100MHz is not due to the reordering itself, but it requires more memory.

If the incoming commands are repacked in the 100MHz domain of the wrapper, then the new commands will have to be passed to the 50MHz domain while already repacked in a different format. In this repacked format the commands can take up to 319 bits for initialization commands. This requires a buffer which is almost 5 times as wide as the buffer in the previous wrapper which was 64 bits wide.

The first wrapper has quite some disadvantages in comparison to the other wrappers in terms of complexity. It will be very hard to alter the core in such that its signals can be send and received at 50 MHz. This alone makes the wrapper less practical to use than

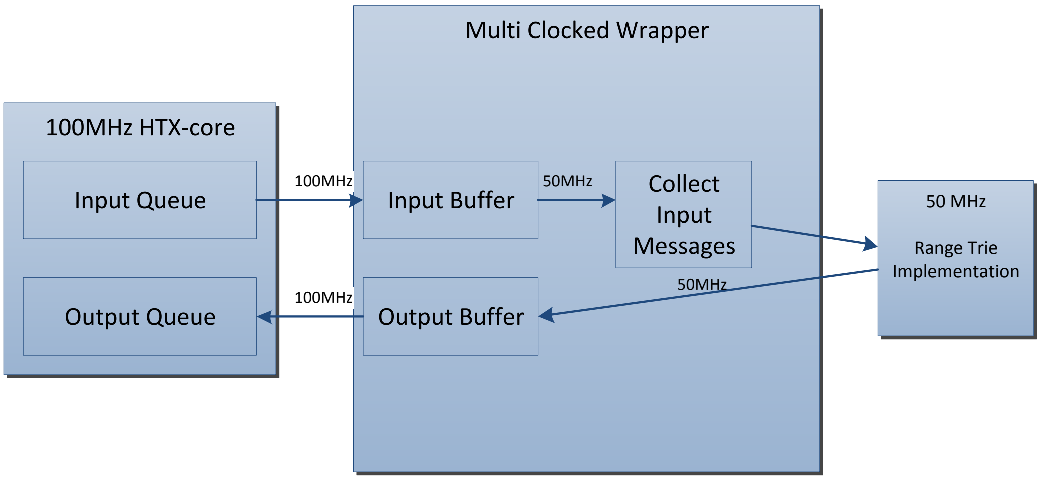


Figure 3.3: *Wrapper Implementation 2: Reordering in 100MHz domain. Wide input buffer, small output buffer.*

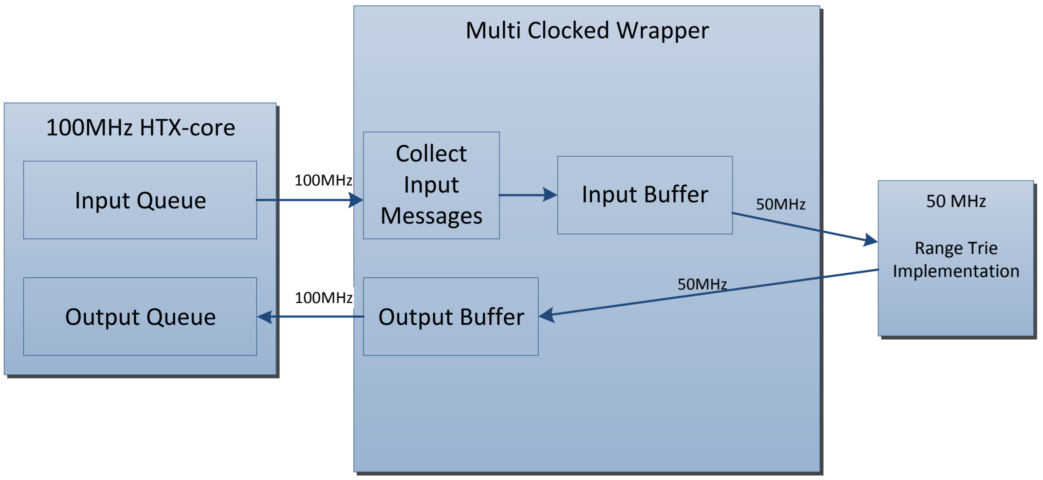


Figure 3.4: *Wrapper Implementation 3: Reordering in 50MHz domain. Small input and output buffer.*

the buffered versions. Even though the total latency will probably be higher due to the introduced buffers.

For a good comparison between the other two wrappers in terms of working frequency and memory usage see Table 3.1. With the third wrapper the work is done in the 100MHz domain and it requires a 319 bits buffer to the 50MHz domain. The second type does the reordering and repacking in the 50MHz domain, but requires only a 64 bits buffer.

Table 3.1: *Comparison for different wrapper implementations, the complexity is defined as how complex would the design be in comparison to the other possible wrappers*

Wrapper	Frequency	Buffers IN	Buffers OUT	Complexity
nr. 1	50MHz	none	none	High complexity due to catching and sending commands to the 100MHz platform.
nr. 2	50MHz	64 bits	64 bits	Lower complexity due to buffering commands.
nr. 3	100MHz	319 bits	64 bits	Lower complexity due to buffering commands.

It worth noting that the most common operations for the Range Trie CCU will be address lookups. In the case of a lookup there will be no need for any preprocessing or repacking and using a very wide buffer of 319 bits to store a 32 bits address is quite some overhead. Therefore the choice has been made to implement wrapper type 2, allowing more memory savings while accepting the longer processing delay for initialization and update commands.

3.3 Range Trie

The first version of the Range Trie design including Longest Prefix Match as delivered by [12] did not work out of the box, therefore modifications and extra functions were necessary. The initial design did have most necessary components, but many of them were not connected or required fixes in the timing. In this section some of these additions and modifications are singled out and explained.

3.3.1 Memory blocks

In the original projects all memories were implemented as lookup tables or LUTs, which will be changed to BRAM memories for this Range Trie implementation. An advantage of an implementation in LUTs is responsiveness or delay as the result of a memory lookup is available the same clock cycle. A downside of LUTs is that they are built out of logic slices while the dedicated memory components (block rams or BRAMs) of the FPGA are not being used.

In this implementation these BRAMs are used while circumventing their downside of needing a clock cycle to fetch the lookup result by early reading. BRAMs do not have this added delay while writing. By using BRAMs with different read and write-address ports it is possible to write a value and already fetch a value for the next cycle at the same time. This means that a write can occur at the same time the (early) read-address is also given so the read-output has the correct value at the same moment as the original Range Trie design. This hides the latency of the BRAM memories and does not require other modification to the design then getting the addresses one cycle earlier then in the original design.

To get the read-address one cycle earlier the required signals are forwarded in the pipeline. For calculated values this could become a problem as the addresses might appear too late in the cycle, or they will alter to early near the end of the cycle. But as many addresses are almost directly received from memory this will not be an issue. For the spare levels the introduction of an extra cycle was necessary, as will be explained in Section 3.3.5.

3.3.2 Memory Utilization

As said in the previous subsection there are quite some memories in the design to store all information about the nodes. In total there are five elements that have to be stored with each node, namely: Comparison values, Control values, Prefix-lengths, Next-Pointers, Prefix-lengths, Action pointers and for each level there is a set of actions. For the spare levels there are no Next-Pointers stored directly at the nodes, but these are received indirectly because of the dynamically allocated memories. In the next sections the different elements are discussed and is explained how they are stored in memory.

3.3.2.1 Comparison and Control values

At every node the given input address has to be compared to determine the correct range. At each level there is a number of comparators and prefix- and suffix-comparators, all these comparators need to have values to compare the input address to, which are called the compare or comparison values. Next to these compare values, the comparators also need to know how the comparators need to be set up and which bits of the input address need to be compared.

In memory, the control and comparison values are stored together as both are required at the same moment. Also, the control values contain relatively few bits per node which makes a specific memory not necessary. In memory the compare values are stored in order of size. In such that the first comparator will compare the input address for the smallest bound and that the last comparator will be used to compare the input address against the largest bound. In case of a shared prefix or suffix (Rule 2 or 3) it is required to separately compare the prefix or suffix. The value associated with this shared part will be the comparison value at the lowest index as is shown in Figure 3.5.

The control values define the setup in which the comparators are utilized and what type of modifications have to be performed on the incoming address before it is send to the comparators. The different control values make it possible for the design to support all five optimizations as defined with the Range Trie in Section 2.3.1. The control values are made out of the following information:

- **shift control**: The incoming address might be shifted before the other modifications take place, this value determines how many shifts of 2bits have to occur.
- **prefix-suffix mask**: As defined by Rule 2 and Rule 3, bounds can have a shared common prefix and suffix which can be ignored while comparing the individual bounds.

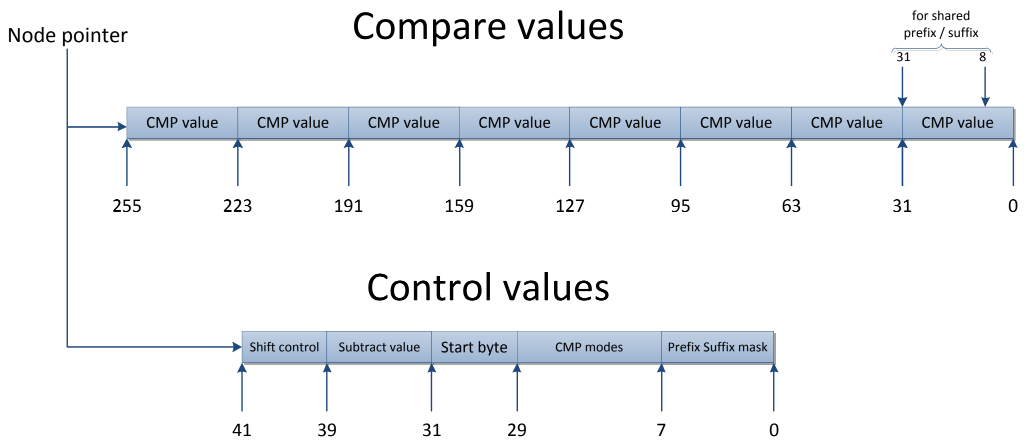


Figure 3.5: *The memory lines for the compare and control values. These are addressed per node and contain the required information for the comparators to perform a lookup on a specific node. The compare values are the values to which the incoming address is compared. The control values determine the comparator setup and what modifications to the incoming address are required.*

- **cmp modes**: The comparators can be setup in different ways, for example to compare 32 bit values or twice 16 bits, or 8 8 16 and 8 8 8 8.
- **start-byte**: The start-byte is used to create the correct input for the comparators if multiple smaller comparisons are performed (for example 8 8 16).
- **subtract value**: With what number should the incoming address be subtracted with before passing to the comparators. As defined by Rule 5.

In Figure 3.5 an example of memory lines with compare and control values is displayed. The comparison memory line consist of 32 bit wide comparisons values and for the control values a line consists of 42 bits which tell the comparators how the compare values should be used and how the result of the comparators should be interpreted.

With the input address, the compare and control values, the comparators determine what the offset is to the next node. This offset has to be added to a base pointer for the next level to get the final address of the next node which has to be visited by the lookup algorithm.

3.3.2.2 Next-Pointers

Nodes in a certain level are stored in order, such that the node with the smallest ranges are at the smallest address. But from the node in the previous level it is unknown how many nodes with smaller ranges exist in the next level, as it just has information about its own compare values and the offset due to these. It is therefore necessary to know from where this offset has to be taken.

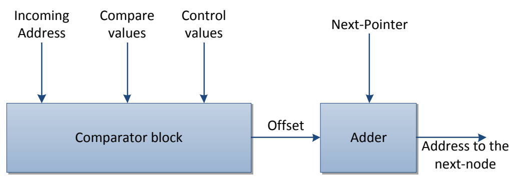


Figure 3.6: *The address for the next visiting node is calculated as the offset determined by the comparators and the next-pointer as stored with the node. Or at the spare levels in the specific pointer memories.*

The next pointers are used together with the offset as determined by the comparators to determine the pointer of the next node to visit. Figure 3.6 shows a sample lookup in a node where the offset is determined and the next node is calculated using the next pointers.

As explained there is only one next-pointer stored for each node in the fixed levels. For the spare part this is different due to the dynamic allocation of memories. At the time a node in the spare part is created it is unknown if there is going to be any next node and no memory is allocated for it. To be more memory efficient there are two possible next-pointers for each node in the spare levels to save memory when trees in the spare part are not very dense. In that case it is possible to allocate memory in the first spare level for only half of the last node in the fixed part.

3.3.2.3 Prefix-Lengths and Action Pointers

For each child that is accessible from a node a prefix-length and action pointer is stored at the parent. The number of children that can be reached is equal to the number of bounds of a node plus one and this is also the number of prefix-lengths and action pointers that have to be stored.

With the action pointer it is possible to select the action corresponding to that prefix. These actions could have been stored directly at the nodes, just like the prefix-lengths. But the actions require more memory than the pointers and it happens that many ranges share equal actions, specifically for short prefix-lengths.

The prefix-lengths are also stored at the parent, just like the action pointer, and is used to perform Longest Prefix Match. The current highest prefix-length and associated action is taken along the pipeline. Whenever a larger prefix is found in a lower level this higher prefix-length and associated action will be forwarded.

Both the action pointers and prefix-lengths are stored in the same way so both can be pointed to in an equal way. In the memory stage in the pipeline both the action pointers and the prefix-lengths are retrieved, as shown in Figure 3.7. The offset, as determined by the comparators, is used to get the correct values from each of the lines.

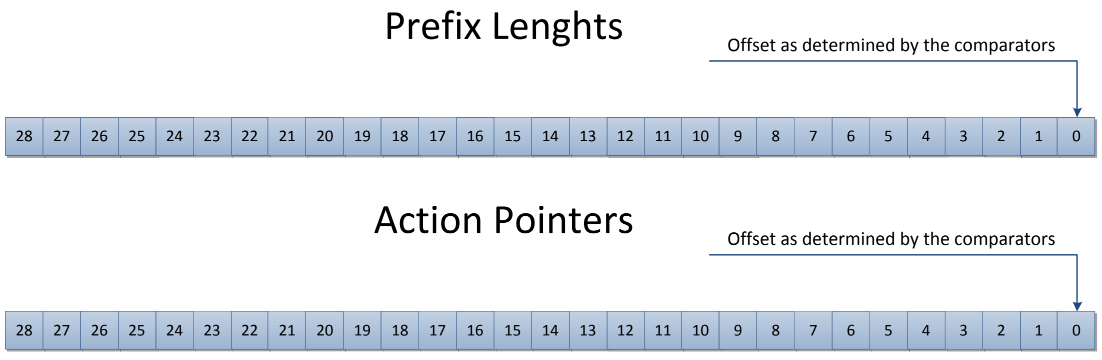


Figure 3.7: *The action pointers and prefix-lengths gives information required for the longest prefix match and are indexed by using the offset as determined by the comparators. When the selected prefix-length is larger than the already known prefix-length in the pipeline, then the action pointer will be used to select the corresponding action. This action and the prefix-length is then given through the pipeline as possible output of a lookup.*

3.3.2.4 Actions

When the comparators have selected a prefix-length this value is compared to the prefix-length as already known from the previous levels. Whenever the just selected prefix-length is larger than the previous, then this one should be given down through the pipeline. With the action pointer which is also selected the new action should be read from the action tables.

When an action is read from the memory it is sent down through the pipeline only if the prefix-length from the current node is larger than the prefix-length that currently travels down the pipeline. This ensures that the prefix-length and associated action at the end of the pipeline satisfies the Longest Prefix Match. Whenever an update occurs a value in the action tables will only be overwritten when the new action holds for all bounds with that action pointer, otherwise a new action pointer will be used and the new action will be stored in an unused location.

3.3.2.5 Summary

With the subject of actions all elements that are stored into the different memories are discussed. In short, each node has a memory line with different compare and control values to control the comparators. These comparators give an offset. This offset together with the next-pointer gives the location of the next node to visit. The offset is also used to select the prefix-length and action pointer of the selected range. Finally the action pointer is used to select the action which is passed down to the output when it satisfies the Longest Prefix Match. The actions will be discussed more elaborately in the next section on their purpose and why they are in hardware.

3.3.3 Actions

The actions are the values which should be returned to the software with every lookup, as this was not yet the case with the initial design, this has been added to the design. The actions tell the software what action has to be performed, either some manipulation to the packet or to which port the packet has to be send. Therefore they are a vital part of the lookup mechanism.

There are several locations where these actions can be stored, for example within the hardware design or in software. Both options have their own advantages and disadvantages. In software it is very easy to store multiple prefixes and their actions to every bound, the hardware would return the bound and the software selects the action associated to the largest prefix for that bound. An update would require the software to add, delete or update a prefix/action pair.

In hardware it is fairly easy to store a single prefix and action for each bound. An update would require the hardware to check the existing prefix with the new prefix and if the new prefix is larger, then update the action. A delete will be harder as there is no history available and in this case the hardware does not know which was the previous largest prefix/action combination for a bound.

During normal operation where most commands are lookups it would be desirable to have the actions in hardware. As it would take the same amount of time as opposed to finding the bound. But when the action is in software an additional lookup in software is required. Having the actions in hardware also has a problem with the history of prefix/actions combinations for each bound. This can be solved by keeping this history in software and when the software detects an update which would require a delete, it gives an update to the old prefix/action combination instead.

3.3.4 Spare levels

Even though the original design contains much of the code and components for the spare levels, the output of the design was connected to the end of the fixed part. This is a clear illustration of how the spare levels were implemented in general. Even after correcting the general implementation they required much work to fully function as already described in [12].

The spare levels are in the design to support the addition of bounds during run-time. After the initial designs of the spare levels were connected they did not seem to support this completely. Only one single bound could be added during an update and the bounds could only be added in order.

It is possible that links in the network fail, this will cause an update for the routing table which actually tells that a certain range should be deleted. Whenever the bounds of the range are in the spare levels, the bounds should be deleted to save memory. With the initial design this was not yet possible. When multiple deletions have occurred it is possible that one of the allocated memory blocks of the spare part is empty, it would be a waste of memory if this memory block could not be used in different subtrees.

In the next sections these three issues with the spare level designs are discussed and their functionalities are implemented. For the out of order and double bound additions this meant that the result of the first update cycle should be forwarded to the input of the second update cycle. As for the bound deletion and the memory recycling, this meant changing the memory management of the spare levels together with the addition of recycle bins for memory blocks.

3.3.4.1 Out of order bound addition and Double bound addition

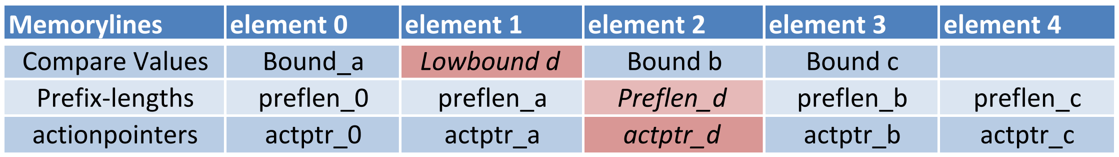
During the implementation of the spare levels it seemed that the spare levels did not work correctly, as only one bound could be added in an update and the bounds had to be in order to maximize memory usage. After implementation of the *double bound additions* and *out of order additions* the spare part works as described in this section and as initially was intended.

As already explained in Section 2.3.5 a subtree will be inserted into the spare part when a certain bound does not exists in the fixed part while it is used during an update. Whenever a subtree under that location in the fixed part already exists the spare part should add the given bound to the first level of the subtree until that node is full. Whenever this happens another level for the subtree should be allocated dynamically and the bound should be inserted into the newly created spare level.

It is important that the second (or possibly larger number) spare level should only be created when all previous spare levels are completely full. As the memory for the spare levels is allocated dynamically it is possible to have quite a large number of subtrees under the fixed part and it is a waste of memory if all those subtrees will be allocating second spare level memories while their first spare levels are not yet full. Therefore it is important to insert bounds into already existing spare levels where possible.

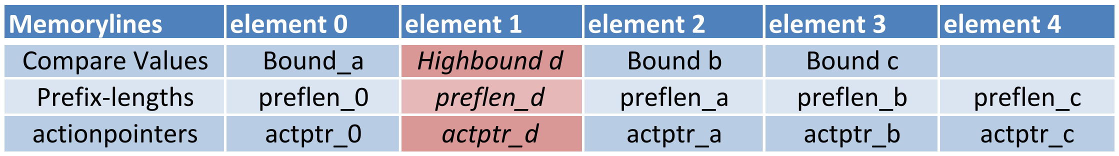
During the insertion of new bounds there are a few issues which have to be taken into account, for example: What values for the prefix-lengths and action pointers should be moved to which position. If a low-bound is inserted at location x, then the prefix-length and action pointer should also be inserted into that location and their previous values should be moved to location x+1. But for a high-bound, the prefix-length and action pointer should be inserted at location x-1 moving that location to position x, see Figure 3.8 and Figure 3.9 for both examples. This comes due to the fact that bounds are from low-bound to high-bound not including the high-bound itself.

As it is possible that an update uses two non-existent bounds the design should support this correctly. It would be a strange property if this would not be supported. In Figure 3.10 a double insertion of bounds occurs within the same update command.



Memorylines	element 0	element 1	element 2	element 3	element 4
Compare Values	Bound_a	*Lowbound d*	Bound b	Bound c	
Prefix-lengths	preflen_0	preflen_a	*Preflen_d*	preflen_b	preflen_c
actionpointers	actptr_0	actptr_a	*actptr_d*	actptr_b	actptr_c

Figure 3.8: *Insertion of a low-bound*



Memorylines	element 0	element 1	element 2	element 3	element 4
Compare Values	Bound_a	*Highbound d*	Bound b	Bound c	
Prefix-lengths	preflen_0	*preflen_d*	preflen_a	preflen_b	preflen_c
actionpointers	actptr_0	*actptr_d*	actptr_a	actptr_b	actptr_c

Figure 3.9: *Insertion of a high-bound*

3.3.4.2 Bound deletion and Memory block recycling

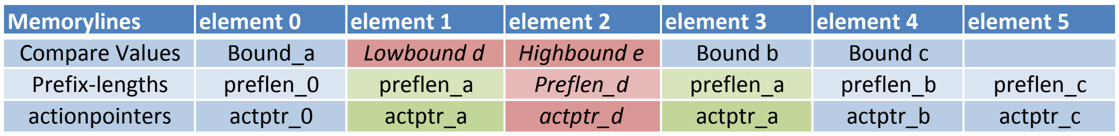
Due to update commands certain bounds get different prefix-lengths and actions. It is also possible to have the deletion of a certain prefix for example due to an outage or the remote host is just non-responding. If multiple of those prefix deletion occurs it is possible to have bounds in the spare levels which are not used anymore.

When after many updates a specific bound in the spare levels is unused, it might be deleted to save memory. Even though this is one of the requirements set by [12] it did not work correctly. For the implementation by this thesis this functionality has been integrated in the designs.

After the deletion of bounds it is possible that one of the dynamically allocated memory blocks will be empty. Whenever this is the case there is no need for the block to stay allocated to a certain subtree. Therefore recycle bins are added to store the recycled memory blocks. Whenever a subtree tries to allocate a new memory block, first the memory blocks from recycle bins are used to save as much memory as possible.

3.3.5 Adding stages in the pipeline

As said in Section 3.3.1 the memories need an extra cycle to fetch memory elements, which is not taken into account with the previous design of the Range Trie. This comes due to the way the spare levels have assigned the different memories. For the spare levels there are memory blocks which can be assigned to each spare level when they require extra memory. These dynamically assigned memories break the default way of



Memorylines	element 0	element 1	element 2	element 3	element 4	element 5
Compare Values	Bound_a	*Lowbound d*	*Highbound e*	Bound b	Bound c	
Prefix-lengths	preflen_0	preflen_a	*Preflen_d*	preflen_a	preflen_b	preflen_c
actionpointers	actptr_0	actptr_a	*actptr_d*	actptr_a	actptr_b	actptr_c

Figure 3.10: *Insertion of double bounds*

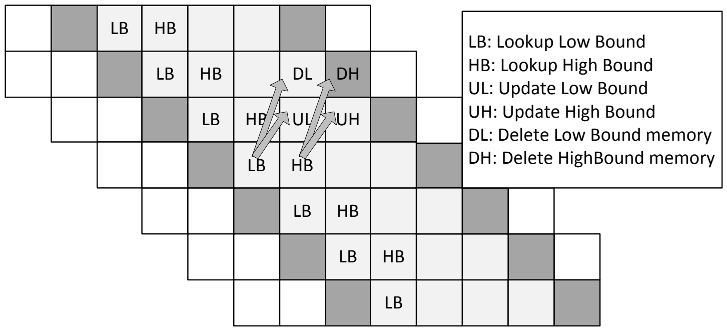


Figure 3.11: *Pipeline while updating, originally there was no room to update the pointer memories*

addressing as used in the fixed levels of the Range Trie. In the fixed part the base address of the children are stored at the parent. But for the spare levels this would not work as the memory would be very sparsely filled, or the parent should be updated whenever a subtree is created.

In the case that the parent is updated, the memory block associated with that address should be large enough to contain all possible children of that node which probably will just be filled very sparsely. In the current implementation it is possible to assign up to 4 smaller memory blocks to a node in the fixed part having the ability to place the memories in a more fine grain manner and have a better utilization of the total memory capacity. The downside of the dynamically assigned memories is that an extra lookup is required to get the correct base address.

This extra lookup causes each spare level to have an extra pipeline stage compared to the fixed level. This extra cycle was not necessary in the design with LUT memories as they would give the correct output immediately and not a cycle after the correct address is given. Although the usage of BRAMs has led to this extra cycle, it does not affect the cycle time of the overall design what could have happened with relatively large LUT memories.

3.3.6 Extra cycles required for updates

The updates initially take a bubble of 4 cycles as explained in 2.3.4, but there is at that moment no possibility to correctly recycle a memory block. Only after the calculation stage it is known if a memory block should be recycled. The pipeline however, already prohibits the update of the pointer-memories after the calculation stage. As can be seen in Figure 3.11 where the block denoted by DH (delete high-bound memory location) falls outside the update bubble. Thus an extra cycle in the update bubble is necessary if the spare part should be able to reuse memory blocks.

In Figure 3.12 the extra cycle is added and the locations where the pointer memories can

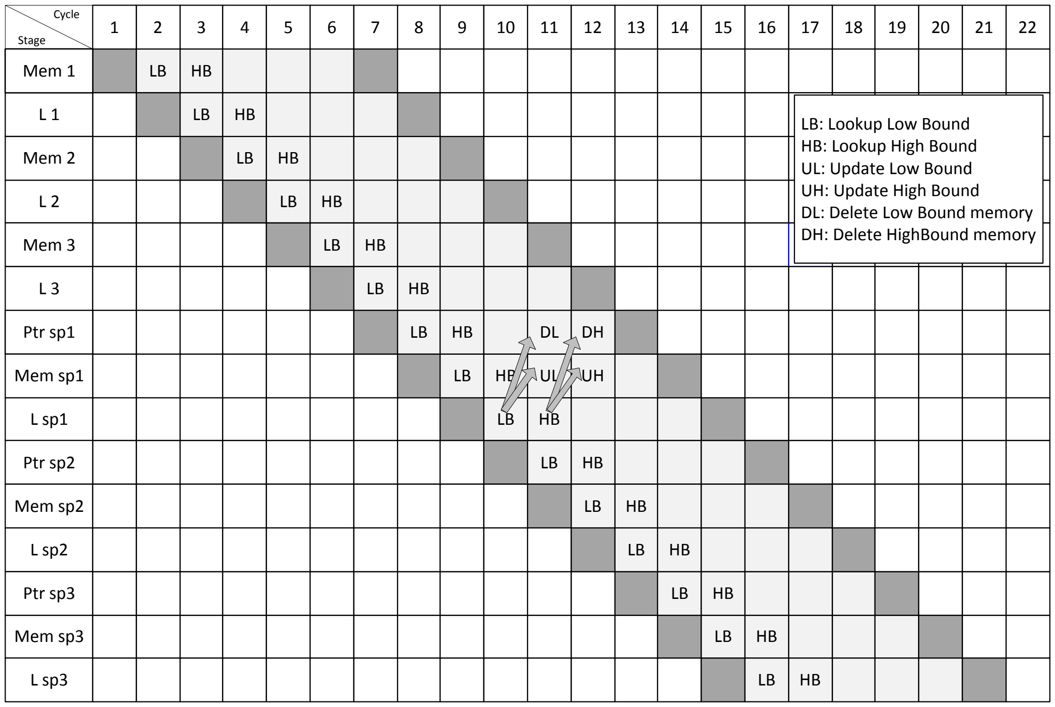


Figure 3.12: *Pipeline while updating, with extra cycle in update bubble to update the pointer memory*

be updated are annotated with DL (delete low-bound memory location) and DH (delete high-bound memory location). When a memory block is empty, thus is not used anymore due to bounds which are removed, should go to the memory recycle bins associated with each spare level. Whenever a certain level requires a new memory block, first the blocks in the recycle bin associated with that level are used.

3.3.7 Retrieval of the tree

As discussed in Section 2.3.5 a downside of the way new bounds are added to the Range Trie is that after a certain amount of bound additions the spare memory can be full or that the all spare levels are used for a certain subtree. In order to prevent this or to solve this issue when one of the two events occur, the subtrees need to be incorporated into the regular and more optimized fixed part of the Range Trie. Rebuilding the Range Trie is a very computational intensive task due to the many optimizations and can therefore not (yet) be efficiently executed by hardware. Thus far this remains a task that has to be performed by the host computer. To feed the software with the correct information all memories need to be readable by software. This functionality was not yet implemented in the previous designs of the Range Trie as no connection to software was available.

As the fixed and spare part are different in implementation, there will also be different commands that the software needs to execute to retrieve information from the different parts. For the fixed levels the following commands are available:

- Get memory contents With this function it is possible to retrieve almost all information available of a certain node, including the compare values for the different child nodes. Other information is the pointer value for the next level, the action pointers, prefix lengths and comparator control values.
- Get Actions The address for the action is calculated with the action-pointer and the base offset from the parent node. The requested action is returned.

For the spare levels it is not possible to send all information back with just a single command as the pointer values are not directly known. An extra lookup for the base address in the next level is therefore necessary to get all required information. This corresponds with the added stage in the pipeline as discussed in Section 3.3.5.

3.3.8 Interface to software

At this moment there is a working version of the Range Trie implemented as CCU in an FPGA connected to a general host computer. In order to use the Range Trie as forwarding engine, the software will have to send the necessary configuration of the trie, the lookups, and possible updates. The total list of commands implemented with this thesis is given in Table 3.2.

The software will be used to build the tree structure for the fixed part of the Range Trie, which is transferred to the hardware using the initialization commands. After the initialization, the hardware is aware of the tree structure but has not yet any information about the action and prefix lengths. The software will therefore send update commands to update every range with the correct prefix lengths and actions. After this batch of updates, the Range Trie design is ready to be used for route lookups.

In Appendix E a full description of the used communication scheme is given. The design does not only use the reads and writes to a software array, but also the exchange registers provided by the implementation of the reconfigurable hardware. The exchange registers are used to transfer basic information like *spare memory low* or *spare level full* is given. The reason for using the exchange registers is that this information should be available from anywhere in the program without delaying the normal execution as forwarding engine.

3.3.9 Pipeline of the Range Trie design

The pipeline of the Range Trie has been altered during the work associated with this thesis. The complete new pipeline is visible in Figure 3.13 and includes all alterations like the feedback from the processing-stage in the spare part to the pointer memories to free the dynamically allocated memories. For each element in the tree its global function or use is described.

Table 3.2: *Commands to the Range Trie hardware implementation*

Command	Description	Expected output
Reset	Reset spare levels so that all used memory blocks in the spare levels are empty and they can be reused cleanly afterwards.	There will be no value returned.
Lookup	This command requires the address to be looked up.	Returns the action and prefix-length (using LPM) and the address.
Initialization	During the initialization the tree structure in the fixed part is built.	The initialization commands do not return any value.
Update	For updating a range of addresses. It requires the low-bound, the high-bound, their prefix-lengths the new action and finally the old prefix-length (mainly for bound deletion).	It returns both bounds including the new associated action.
Get memory contents	it requires the level of which the memory should be read. It also requires the address of a bound.	The memory contents of a certain node. It includes compare values, pointer values, action pointer and control values.
Get actions	Get the specific action at a certain location. Requires the address next to the level.	Returns just the action.
Get spare level pointers	Retrieve a base address for a spare level. It requires level and address.	The base pointer.
Get spare level memories	To retrieve node information from the spare levels. It requires a level and node address.	It returns the compare values, action pointers and control values. The pointer values are retrieved from the pointer blocks.
Get spare level actions	Together with the action pointers from the memory contents the action address can be calculated. This address as well as the level is required as input.	It returns the action located at the given location.
Get memory assignment	No extra input is required.	Returns an array of 2 bit values. These values tell which of the memory blocks in the spare levels are allocated and to which level.

The pipeline itself is build up in a general and regular fashion in such that the addition of extra levels can be done without addition of the other parts. Just the memory sizes need to be resized as lower levels in the fixed part require more memory as they can support more nodes that levels earlier in the pipeline. Other research is currently done on the effect of having more levels in the tree and having duplicate levels to better spread possible lookup load in order to cope with a high demand circumstances.

Due to the clear differentiation between the different levels and even between the fixed and spare part makes this design very scalable in terms of available resources. The spare part can be extended to use almost any amount of memories. Although it might not be worthwhile to assign too much memories to the spare part, as subtrees only accept three levels with nodes and the number of levels should be increased to a higher number.

3.4 Summary

During the work for this thesis many errors in the first version of the Range Trie design are repaired and new functionalities are added. Some required adjustments have been made to the Range Trie design, for example the addition of extra update cycles and the extra pipeline stages so the BRAM memories could be used, but also adding the complete spare part to the pipeline and allow the design to perform updates. While correcting these issues, also the design was incorporated into the HTX platform and the communication scheme is designed.

Next to necessary additions to get the basic Range Trie working, also some improvements have been incorporated. For example the possibility for the software to read the tree information back so it can be rebuild and further optimized. Together with the HTX platform a functional prototype of the Range Trie can be used with software support. The next step is to evaluate its performance.

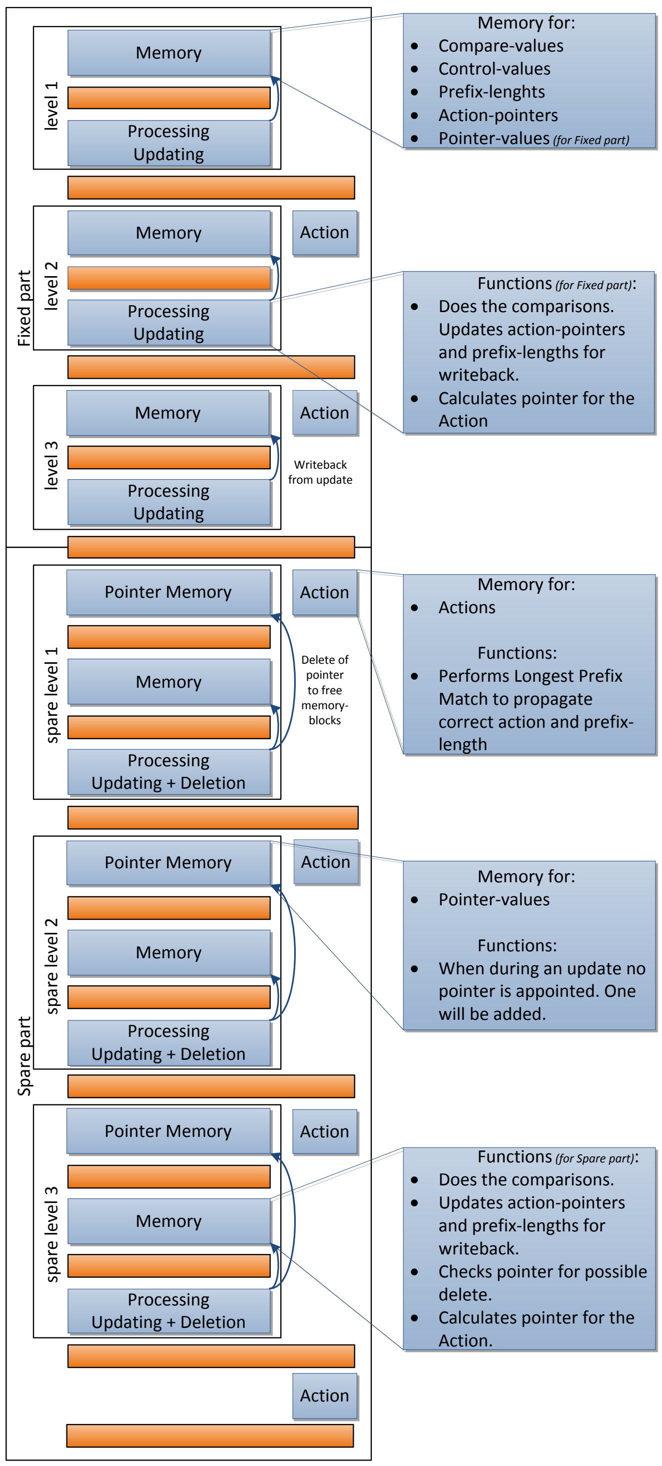


Figure 3.13: *Pipeline of the Range Trie design*

4 Evaluation

During the work for this thesis, a fully functional design of the Range Trie is implemented on an FPGA and is interfaced to software using the HTX Reconfigurable platform. With this implementation the Range Trie can be evaluated by comparing its performance to the default IP Lookup algorithm of the Linux kernel. Both the Linux and Range Trie routing tables will be filled with some default values which will be used during the lookups and updates. Comparisons will vary in the amount of lookups and updates, and both single and batch commands will be send. Due to the last differentiation it will be possible to eliminate the delay due to initialization of the connections.

Section 4.1 and 4.2 describe the measurement setups for both the software and hardware. For the software two different methods to initiate a route lookup are explained. Firstly, the setups are used to measure the lookup latencies in Section 4.3 with different amounts of lookups. Next to just lookups, also combinations of lookups and updates are measured. In Section 4.4 the impact of the updates to the performance of the Range Trie is discussed.

The Range Trie supports multiple types of commands, for each of the commands the individual latency is determined in Section 4.5. From these latencies it can be seen that the commands which return more data are generally slower than the commands which return little information.

During the different lookup measurements, there seemed to be an additional delay which occurred in a few cases. Section 4.6 investigates whether these additional delays occur on a regular basis by executing the same testbench many times and comparing the individual execution times. Section 4.7 describes the resource utilization of the Range Trie implementation on the Virtex-4 FPGA.

4.1 Measuring performance in software

The operating system of the host is Gentoo Linux (Kernel 2.6.29-gentoo-r5) which includes the IPROUTE2 package. This package is used to populate the routing tables before the measurements. For a complete description of the IPROUTE2 package please see Appendix A. The IPROUTE2 package contains a method to lookup the next hop for a given destination by performing a routing table lookup directly from the kernel.

The execution time of a lookup using IPROUTE2 was quite long (about 3 seconds for 1000 lookups). Further investigation showed that the command *iproute get IPADDRESS* can be equivalent to sending a ping to the destination without actually sending any packets, please see the documentation [14] Section 9.5.7. This means that the measured execution time was not just the time it takes to perform a route lookup but might include other operations.

As this is not what should be measured, another method to initiate a lookup is required. One of the methods to manually run the Linux routing table lookup (the FIB_LOOKUP kernel function) is to use NETLINK, which is also the method IPROUTE2 uses internally to communicate with the Linux kernel [8][7].

NETLINK is a method to communicate with the Linux kernel using sockets. It supports various operations and message types, one of which is NETLINK_FIB_LOOKUP, which does exactly the necessary operation. Unfortunately there is almost no documentation about this operation, opposed to almost every other NETLINK operation. There is a library, *libnl*, which makes the usage of the NETLINK interface easier by adding some functions to complete and convert messages. The libnl library has an extensive example database including one which does route lookups [10], which works almost right out of the box. In Appendix B the example code is attached.

The time needed for a route lookup using libnl and NETLINK is in the order as expected and will thus be used during the comparison against the time needed for a route lookup using the Range Trie hardware implementation.

4.2 Measuring performance in hardware

For the Range Trie there is a program has been created which has the possibilities to send the required command for initializations, as well as performing lookups, updates and reading back the response of the hardware. The code for the program is attached in Appendix C.2 and C.3.

Part of this code can already be used as base for the creation of a Linux kernel module that overwrites the current routing table lookups. This way the Range Trie will take the place of the normal routing table. As a reminder; not only should there be a read module, but also one for updating and deleting routes from the table.

For the evaluation and comparison of the Range Trie with the Linux kernel there will be a differentiation between only lookup requests and a combination of lookups and updates. For both methods there will be a differentiation between batch-wise operations in which the operations will be started in one large batch without the overhead of setting up the sockets, and in single operation where each operation is individually executed by the function.

4.3 Measurements: Lookups

In Table 4.1 the different timings for the different route lookup methods are given for various amounts of lookups. More timing measurements are attached in Appendix D. For each method also the throughput is determined by executing a batch of 50 million lookups.

It seems that even for small number of lookups the hardware version is already around 10% faster than the default Linux routing lookup mechanism, for both the batch and individual lookups. The duration of the single hardware lookups stay close to the duration of the software lookups when more lookups are executed. The difference is just a few milliseconds for 1000 lookups. It is remarkable that the batch lookups in hardware are executed twice as fast as the software variant. For 1000 lookups the software variant is twice as slow as the Range Trie software implementation.

Tests with more lookups have also been performed up to 50 million lookups in a single batch. The execution of 50 million lookups in hardware takes slightly longer than 11 seconds (11.33), which results in a throughput of 4.411M lookups/second on average at a 50MHz clock. When the same batch is run in software this takes almost 9 minutes (8:51) which results in a throughput of 94K lookups/second. The throughput for the individual commands has not been measured as 50 million lookups would result in very high execution times.

In Figure 4.1 and 4.2 the data of Table 4.1 is displayed in graphs. The difference between the batch and individual operations is substantial. Apparently a significant amount of time is spent in setting up the connections and waiting for the final results. In the batch operations the connections are made only once and large quantities of commands are send through the already existing tunnel. The batch operations therefore gives a better insight in the functions themselves.

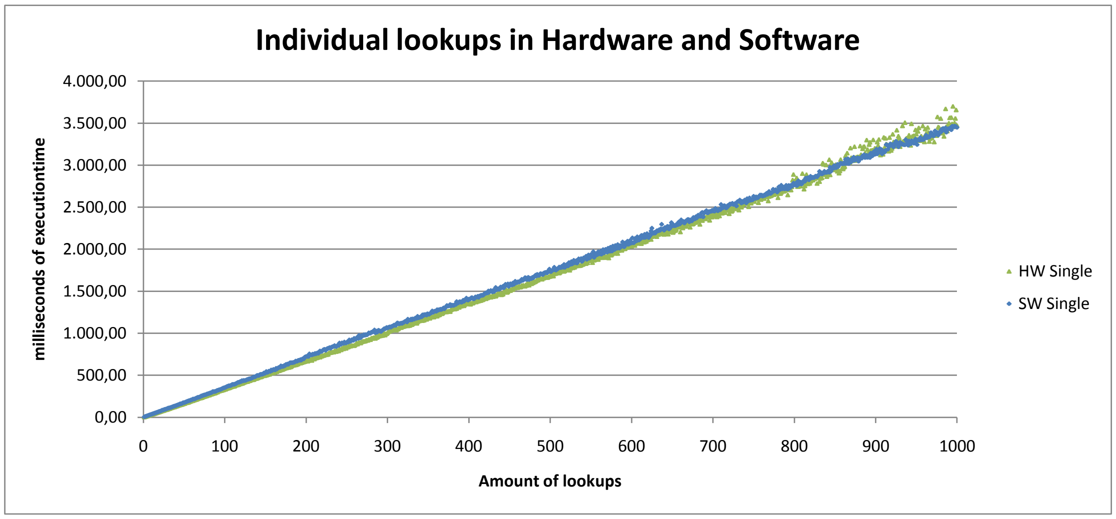


Figure 4.1: *Execution times of individual commands*

In few cases the hardware lookups have a much higher execution time than expected, as can be seen in Figure 4.2. The hardware probably has to wait for a memory page to be swapped from disk to the main memory before it can access the commands to be executed. This is one of the downsides of the HTX platform as it currently is not able to stream data directly to the accelerated function as is the case for the current Linux routing table implementation.

Table 4.1: *Timings (in microseconds) for the three lookup methods for different amount of lookups and the throughput for each method for longer execution times with 50 million commands.*

Lookups	Batch			Individual		
	Range Trie	Software	Speedup	Range Trie	Software	Speedup
1	3.62	3.83	1.06	4.75	5.00	1.05
10	3.72	3.93	1.05	34.08	35.91	1.05
25	3.70	4.12	1.12	82.35	89.18	1.08
50	3.93	4.42	1.10	165.47	173.71	1.05
75	3.81	4.66	1.22	248.39	265.57	1.07
100	3.78	4.90	1.29	340.69	353.33	1.04
150	3.95	5.60	1.41	505.38	529.11	1.05
200	3.91	6.22	1.59	673.28	722.15	1.07
250	4.32	6.72	1.56	840.92	893.01	1.06
500	4.85	9.16	1.88	1676.76	1733.78	1.03
750	5.29	11.88	2.25	2587.42	2627.81	1.02
1,000	5.78	13.93	2.41	3471.82	3447.06	0.99
2,000	5.88	26.67	4.54			
5,000	8.37	61.38	7.33			
10,000	13.05	125.13	9.59			
20,000	22.64	253.15	11.18			
50,000	45.01	546.56	12.14			
100,000	46.28	1029.41	22.24			
200,000	90.47	2040.39	27,54			
500,000	184.02	5067.89	2.64			
1,000,000	330.85	11235.15	33.95			
2,000,000	566.62	20166.33	35.59			
5,000,000	1234.61	55249.89	44.75			
Throughput	4,411,060	94,013	46.91			

4.4 Measurements: Lookups and Updates

When a router is operating, lookups are not the only actions it performs. Another action which the Range Trie will have to execute on a regular basis are updates. To give an indication of the performance of the Range Trie implementation with updates, test benches in which 1% and 10% of the commands are updates are executed with different batch sizes. As it is difficult to have 1% updates with 10 or 25 commands, larger batches will be used. In Table 4.2 the execution times are given, as can be seen the jobs with 10% updates have a slightly higher execution time than the jobs with 1% updates. But the difference is almost neglectable for this amount of commands. The fact that the differences are relatively small can also is illustrated in Figure 4.3, where the execution times of either just lookups, lookups with 1% updates and updates with 10% updates are plotted together.

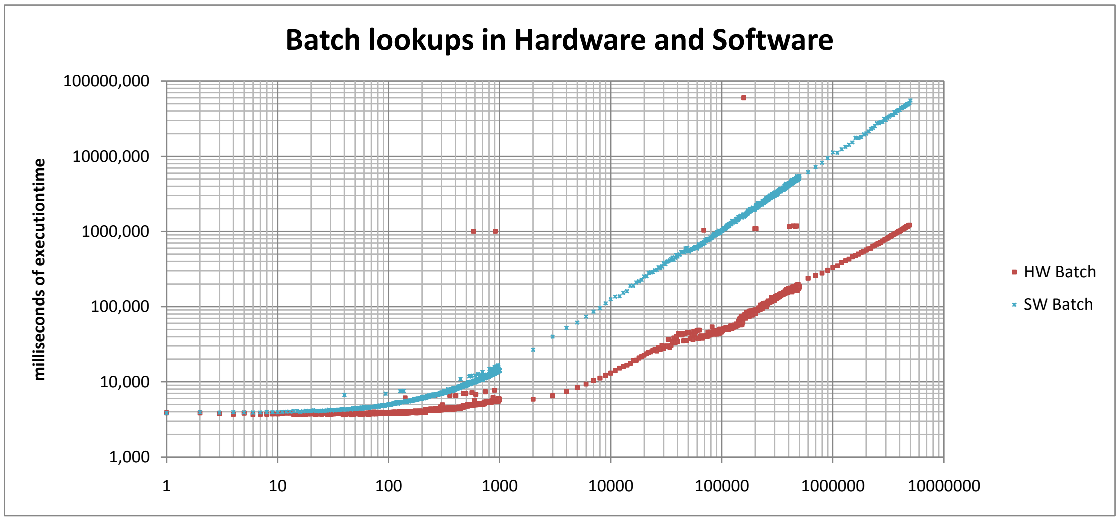


Figure 4.2: *Execution times of batch commands*

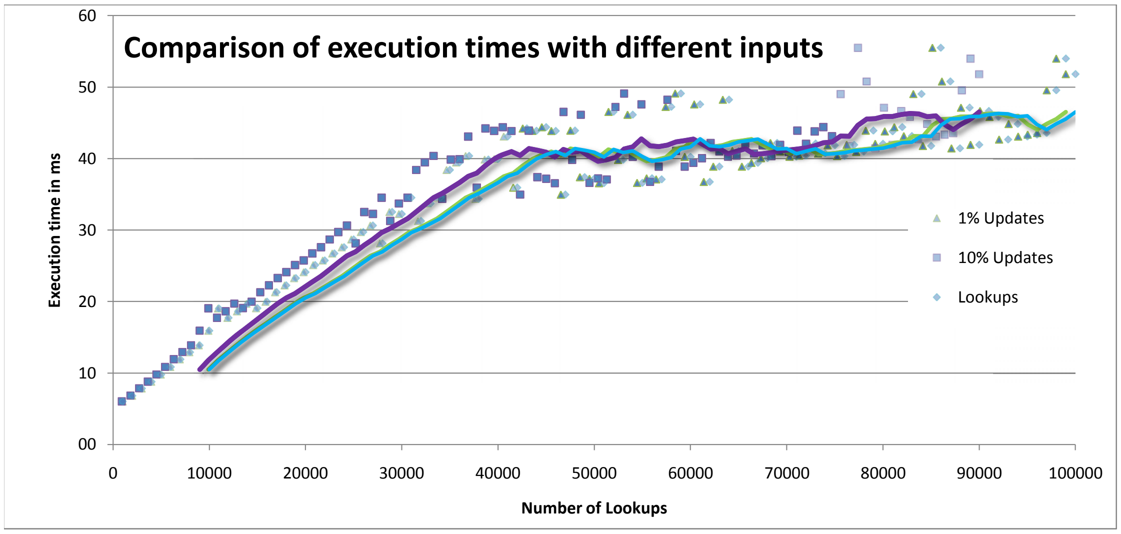


Figure 4.3: *Comparison of execution times differentiated on input type. The input is either, just lookups, lookups with 1% updates and lookups with 10% updates.*

4.5 Latencies of the different Commands

The latency of a command is the complete execution time of that command including the overhead of the communication. Most of the time the throughput can be seen as the most important measure, but it is more scalable with the clock speed which can be increased by pipelining the processing stage of the Range Trie. The latency is therefore also an important measure as this will not decrease as easily with an increasing clock speed. For the most common commands the latencies are described in the coming sections. Table 4.3 contains the list of the individual latencies.

Table 4.2: *Timings (in microseconds) for different amount of commands of with 1% and 10% updates*

Commands	Range Trie batch with just Lookups	Range Trie batch with 1% Updates	Range Trie batch with 10% Updates
1000	5.88	6,03	6,67
2000	6.52	6.84	6.67
3000	7.47	7.86	7.56
4000	8.37	8.8	8.66
5000	9.33	9.80	9.54
6000	10.34	10.86	10.64
7000	11.19	11.93	11.47
8000	12.22	12.93	12.48
9000	13.05	13.89	13.50
10000	14.05	15.93	14.48
20000	23.54	24.12	24.20
30000	29.42	32.25	31.36
40000	44.07	39.93	31.95
50000	45.18	37.18	35.34
60000	48.82	40.25	37.06
70000	40.02	41.01	42.69
80000	43.12	42.06	45.91
90000	45.11	41.93	42.37
100000	53.76	51.81	58.321
Throughput	4,411,060	4,386,187	4,268,111

Table 4.3: *latencies of different commands in milliseconds*

Command	Latency
Init	5.88
Lookup	5.07
Update	6.19
Read fixed level node	7.16
Read spare pointer	5.06
Read spare level node	6.05
Read action	4.78

The commands of which the latencies will be inspected are: initialization, lookups, updates and the different commands to read the tree from the hardware back to the software for a rebuild. Especially for the read commands the latencies are very important as many of the commands rely on each other. So that might not be possible to read the next part without information from an earlier read command. For example, it is not possible to know the address of the actions for a specific node without knowing the action pointers for that node.

4.5.1 Latency of a Lookup

A lookup is the most common action for the Range Trie and also has the easiest interface to the hardware as it only requires the address which has to be looked up. Internally a lookup is quite difficult as the hardware will traverse the tree and at every node the given address will have to be compared. The answer of the comparators will be an offset from the next level. A pointer to the next level is stored and together with the offset the pointer to the next node is determined.

Together with the next node calculation, also the prefix lengths are checked to support Longest Prefix Match. Whenever a visited node has a longer prefix, then this one will be taken along the pipeline together with the associated action. The action and prefix length are both returned as result from the lookup. The latency for this command is given in Table 4.3. The lookup command has one of the lowest latencies, which is perfect as the lookup will be the command that will be executed most.

4.5.2 Latency of an Update

A prefix update command is the most complex command as it can be a normal update of existing prefixes, but it can also invoke an addition or deletion of one or two bounds of the prefix. The hardware first does a lookup on both bounds given by the update command. In the two cycles next to the two lookups the actual update is performed including the possible insertion or deletion of bounds. In the fifth update cycle memory blocks in the spare part which are emptied due to the deletion of a bound are freed so they can be reused.

The latency of an update command will be larger than that of a lookup due to the different lookup and update cycles it will invoke in hardware. This corresponds to the measured latency in Table 4.3 where it can be seen that latency of an update is about 20% higher than the latency of a lookup.

4.5.3 Latencies for Read commands

The read of a single element has an equal latency as a single lookup, but for a full read there are multiple elements required. For the read of a node in the fixed part two commands are required. With the first one the compare and control values as well as the prefix lengths and action pointers of a single node is retrieved. Extra commands are required to get each action which is associated with a prefix. An overview of the latencies is given in Table 4.3. The total time required to read back a complete node in the fixed levels would be the latency to get a node plus the time to retrieve the number of actions associated with that node.

In the spare part more commands are required as the pointer to a next node is not stored at the parent but in the pointer memories. This means that first a read from the pointer

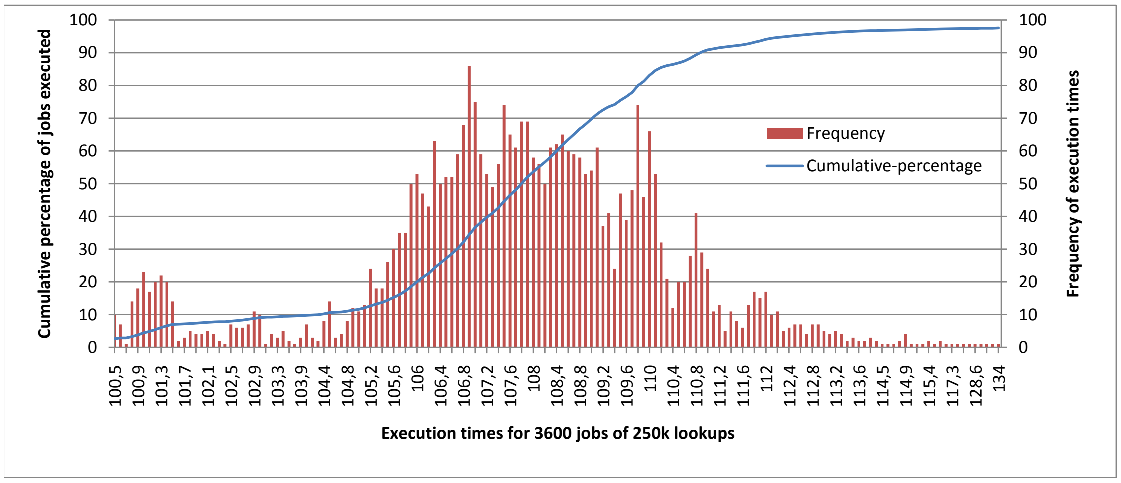


Figure 4.4: *Continuity of the execution times in milliseconds shown as the histogram of the different execution times for all 250k lookup batches.*

tables is required before the node can be read. This means that the total time to read back a node from the spare part would consist of a read from the pointer memories, the read of the node itself plus the read of the individual actions associated with that node.

4.6 Continuity

Continuity of the design can be of great importance, certainly when it has to operate in something as vital to the world as the Internet. The HTX platform has to read the commands from the main memory where they are placed by the software. It can happen that the memory location of the requested page is currently not in main memory, it is swapped to the hard disk by the operating system.

The HTX platform handles this issue by asking the operating system to swap the required data to the main memory. Even though this fixes the issue, it does introduce a fixed delay which can occur randomly with each batch.

In the measurements in Figure 4.2 some measurements already have a much higher runtime than the average. To test if this occurs many times or if this are merely incidents many batches of equal size are timed. The execution times and their occurrences are displayed as a histogram in Figure 4.4. In the histogram it is clearly visible that 50% of the jobs of 250k lookups finished within 107.8 ms with a total average of 107.8 ms and that even though high execution times do occur quite often, there is no distinct second peak at a higher execution time which could correspond to a random additional delay such as a page swap.

4.7 Hardware Implementation

The Range Trie is implemented together with the required designs for HTX platform on an Virtex-4 (xc4vfx100-11ff1152) [9]. The synthesized designs use most of the resources available on the FPGA as is displayed in Figure 4.5. This current design can support up to 256 leaf nodes with each up to 28 bounds in the fixed part and has 16 memory blocks which can contain 512 nodes each in the spare part.

To support larger trees in the fixed part an extra level would have to be implemented. This would not only require memories but also extra slices and LUTs to hold the supporting logic. As all of these resources are very limited larger trees will probably not fit in this device. It would be possible to add some extra memory blocks to the spare levels such that more subtrees can be created, although the limiting factor will again be the number of available levels.

Resource	Available	Usage	
LUTs	84,352	70,082	83%
as logic		65,121	
as route-thru		604	
as memory		1,518	
as shift-registers		2,839	
Block RAM	376	326	86%
DCMs	12	3	25%
Slices	42,176	38,140	90%

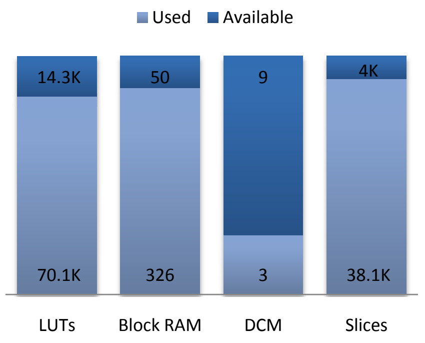


Figure 4.5: *Resource utilization of the design after implementation on the Virtex-4 FPGA.*

The Range Trie implementation already uses a large amount of resources from the FPGA as can be seen in Figure 4.5. Due to this resource usage the synthesizing, mapping and routing of the design takes quite some time with about 75% chance that it will not meet the timing requirements. Mapping and placing with ISE 10.1 only allows the usage of a single CPU core unlike the newer versions of ISE. However, the FPGA is not supported by newer versions which support multi-core mapping and placing. In newer FPGAs (like the Virtex-7) the amount of BRAMs and other resources have increased almost exponentially with high-end models which have more than 300k slices, 46k BRAMs and almost 2M LUTs. Choosing to replace the current FPGA for such a model can ease future development and research on the Range Trie.

4.8 Summary

Different measurements setups are used to determine the performance, individual and batch. First only lookup operations were compared for the different methods in Section 4.3. From this it was clear that batch operations are much faster due to the overhead of setting up the connections. With batch jobs the hardware was already twice as fast than software for a batch of 1000 lookups and with larger batches even the difference is even larger. For jobs with 1% or 10% updates, which are discussed in Section 4.4, there is an increase in execution time, but the difference compared to just lookups is as expected. The total throughput for hardware is 4.4M lookups/second for just lookups, 4.3M lookups/second for 1% updates and 4.2M lookups/second for 10% updates.

Next to the performance measurements over longer periods, Section 4.5 is dedicated to the measured latencies for the most commonly used commands. Even though the overhead of setting up the connection is also taken into account, this measure is still a good indication to see which operation has the highest execution time. From this it is possible to say that any operation that return multiple elements will have a larger latency. The commands that return single elements, like lookups have a low latency of 5 milliseconds. From these 5 milliseconds, most time is in the initialization of the connections as 1000 lookups have a latency of only 5.78 milliseconds.

From Figure 4.2 it was thought that higher execution times could occur to page swaps in memory. To check if this is really happening a check for the continuity was performed in Section 4.6. From the histogram it was visible that no second peak was found meaning that there is no fixed delay which occurs regularly.

Next to timing also the design is checked and discussed in Section 4.7. From the FPGA utilization numbers in Figure 4.5 it is clear that extra levels in the tree can not be implemented in this FPGA as 90% of the logic slices are already in use. Some extra memory blocks can still be added to the spare part as 86% of the BRAMs are currently in use, but without extra levels this probably will not have a great influence on the total design. Currently the Range Trie operates at a clock frequency 50MHz. Higher throughputs are possible if the design is able to operate on a higher clock frequency.

5 Conclusion

Due to the ever growing Internet and the coming switch from IPv4 to IPv6, the route lookup mechanisms in routers have become a bottleneck in terms of delay, a delay which brings down the total speed of the Internet. The growing Internet with evenly growing number of hosts, routers and links will increase the number of entries in routing tables. Together with the fact that the addresses will increase from 32 to 128 bits in IPv6, this will again cause more problems for the routing lookup mechanisms as larger addresses will have to be compared to find the best route. Current mechanisms have started to lag behind as they do not seem to be scalable enough for the increase in address amounts as well as the increase in address lengths. Due to this problem the Internet will become slower in the near future, therefore a solution has to be found.

As in most cases there are different options to solve this problem. One possibility is to implement new mechanisms which scales better to both variables. It is also possible to speedup the current methods by implementing it in different platforms. In such that an algorithm implemented in software, FPGA and ASIC perform different. The trade-off between the three implementation platforms is cost vs performance. With each faster implementation the initial cost of development will increase.

As there are multiple options to find a solution to the problem a trade-off between cost, scalability and usability is required. In this thesis the first implementation of the Range Trie is proposed in an HTX reconfigurable platform which supports function acceleration in hardware using an FPGA. Thus having the best of all worlds, a new and promising algorithm implemented in an FPGA which can be a good starting point to solve the problem.

In Section 5.1 a summary of the work and issues encountered during the implementation of the hardware implementation of the Range Trie will be given. The initial goals and contributions for this thesis will be presented in Section 5.2 and will also discuss to what level they are achieved. Section 5.3 will conclude this thesis by providing some further investigation what could be interesting regarding the Range Trie and possible future work.

5.1 Summary

This section summarizes the basic information as given in the different chapters of this thesis. Starting with Chapter 2, which contains background information about routing tables and their implementations. Two current routing table implementations which use tree structures are discussed. Next to the basic tree structures, it is also possible to use

Range Trees, which use a different method to perform a lookup. The Range Trie is an optimized version of the Range Tree.

From the different discussed methods the Range Trie is most promising to solve the scalability problem which is the main problem for current IP Lookup algorithms and which is threatening the Internet. The Range Trie has five optimizations to reduce the number of bits with each comparison. To implement the different optimizations the tree associated with the Range Trie will be build in software as it is a very complicated algorithm.

The Range Trie is implemented in an HTX environment with support for hardware accelerated functions and the Range Trie is the accelerated function. **For correct functioning within the HTX platform, the platform had to be adjusted and a communication scheme is designed.** The used solution supports communication between the hardware and the host in such that the host can call the accelerated function to perform a specific command, and the hardware is able to return an answer to the command.

During the implementation of the Range Trie, it seemed that the first design did not work as expected. **Many modifications are made and various improvements were done to the design in order to make it fully functional.** In Chapter 3 the implementation of the Range Trie into the HTX reconfigurable platform is discussed. Including the adjustments to the platform, the communication scheme to the software and most of the adjustments made to the Range Trie design. **Some of the modifications on the Range Trie design were performed to implement the design in an FPGA.** The design would not have functioned in the Virtex-4 FPGA without these modifications, for example the switch from LUT memories to BRAMs.

After the implementation of the Range Trie in the HTX environment, **the performance of the implementation is evaluated** an compared to that of the Linux routing mechanism in Chapter 4. During the evaluation some different test sets have been executed to see how the hardware behaves with different types of input. Also the latencies and throughputs for different commands have been investigated.

The question with this proposed implementation is, **will the Range Trie implemented in an FPGA be a possible solution to the problem of scalability?** The Range Trie hardware implementation has been compared to the Linux default routing table and has a throughput which is four times higher than the software variant. This throughput of 4.4M lookups/second is achieved while running at 50MHz which could be increased by pipelining the processing stages. Using a different FPGA or using an ASIC might increase the operating frequency as it is currently very hard for the synthesizer, mapper and router to fit everything on the device. With a higher operating frequency the throughput will further increase, which makes this implementation certainly a field of interest for current day routers.

5.2 Contributions

At the start of this thesis the final goals and contribution were announced, with as main goals: (1) complete the Range Trie design, (2) FPGA prototype the Range Trie design, (3) implement the design in the HTX platform and (4) evaluate the performance in order to determine if the proposed method is interesting for possible implementation in routers. In the following sections these goals will be discussed to check whether they are achieved or to which level.

5.2.1 Complete the Range Trie design

The most important goal for this thesis was to deliver a working design of the Range Trie and check how it performs in hardware compared to software. This required a fully functional design of the Range Trie. The original design was not working correctly for some functions, while other functions were not yet supported. The current design is fully functional and is easily simulated with or without the wrapper.

In order to get the design working correctly, the spare levels had to be implemented completely as well as most of the update functionalities in the fixed part. In the design of the spare part there are a few specific things that are implemented such as reutilization of memory blocks. Memory blocks which are emptied during updates can be reused in other subtrees as they are being freed. In order to include this without many modifications to the pipeline, the update bubble has been expanded from four to five cycles.

5.2.2 FPGA Implementation

Having a design which can be simulated is unfortunately not yet the same as having a design work in an FPGA. During the creation of the FPGA implementation the used memories had to be changed from LUTs to BRAMs. Even though the Range Trie is memory efficient, still quite some memories are required to fit larger trees. As BRAMs are mapped into specific memory blocks they do not use logic slices which now can be used for logic which did not fit when using LUTs. This change required changing the pipeline as BRAM memories respond different then LUT memories.

It is possible to read back the contents of the memories. This is required as the tree will have to be rebuild to incorporate the spare part into the fixed levels. The reading of the memories can be a time consuming task as multiple reads per node are required.

5.2.3 Implementation in the HTX platform

The HTX platform uses a special interface to support the communication between host and hardware and to control the hardware from software. A wrapper has been designed which is the communication layer between the HTX platform and the Range Trie design. The wrapper reads the signals from the HTX platform, requests commands and writes their results.

As the HTX platform expected a function which was able to run at a clock frequency of 100MHz, this had to be changed as the Range Trie design currently runs at 50MHz. Therefore the clock management of the HTX core is adjusted to support a 50MHz accelerated function. As the core of the HTX platform itself does run at 100MHz, the wrapper has to be able to support both 100MHz at the HTX side and 50MHz for the communication to the Range Trie.

5.2.4 Performance evaluation

During the evaluation, the hardware implementation of the Range Trie is compared to the routing mechanism as implemented in the Linux kernel. Both methods are evaluated using batch commands and individual commands. Timing measurements of the individual commands are relative close to each other. The measurements with batch commands gave much more interesting results as the hardware implementation already is twice as fast as the software version for 1000 lookups.

The throughput of the hardware implementation is 4.4M lookups/second which is more than fourty times higher than the throughput of the software method which has a throughput of 94K lookups/second. Both throughputs were determined by measuring the execution time of batch with 50 million lookups to minimize the initial overhead of setting up the connections.

Next to only lookups, also tests have been performed with 1% or 10% updates. With throughputs of 4.3M and 4.2M commands/second respectively this difference is not as large as one might expect as updates use five times the amount of cycles as lookups do in hardware.

All of the goals as set in the beginning of this thesis are met, and even some improvements to the Range Trie design have been implemented such as the freeing of memory blocks and the ability to add multiple bounds within the same update command. Regarding that the hardware implementation is currently operating at a clock frequency of 50MHz, the proposed system seems a very interesting field of investigation for the implementation in a router while solving the scalability issues current routers suffer from.

5.3 Future Work

As this is a first implementation of the Range Trie in hardware and one of the first designs that uses the HTX platform there are some optimizations and future work that might increase the usability of the HTX platform and that might increase the performance of the Range Trie.

- **Allow software to stream data to the hardware as addition to both the driver and the designs of the HTX platform.** Currently the hardware function, or in this case the wrapper, has to request the commands from software. When there is a method such that the software can push the commands in a streaming fashion, then the hardware will not have to wait for the function call, the data request and the arrival or the commands. Such a function would also make the HTX platform more interesting for other streaming or real-time applications.

- **Create a Linux Kernel Module for this hardware.** If the code for the Range Trie is changed into a Kernel Module it might be possible to build a kernel which actually uses the Range Trie as routing table. With the Range Trie as functional routing table it is possible to make much better measurements with respect to throughput and other performance measures.
- **Pipeline the processing stage to increase the operating frequency of the Range Trie.** I personally think that if there would be a new pipeline stage after the compare/decode unit (at least in fixed level 3 and in the spare levels) that it would already be possible to increase the frequency.
- **Create a software version of the Range Trie.** When there is a software version of the Range Trie it would be possible to see whether the speedup as measured in this thesis is due to the change of the algorithm, or by the FPGA implementation.

Bibliography

[1] L. Allison, *Patricia*, http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/PATRICIA/, 1999.

[2] Raffaele Bolla and Roberto Bruschi, *The ip lookup mechanism in a linux software router: Performance evaluation and optimizations*, Proc. of the 2007 IEEE Workshop on High Performance Switching and Routing (HPSR 2007, 2007.

[3] A.A.C. Brandon, *General purpose computing with reconfigurable acceleration*, Master's thesis, TU Delft, November 2010.

[4] HyperTransport Consortium, http://www.hypertransport.org/.

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

[6] R. de Smet, *Range trie heuristics for variable-size address region lookup*, Master's thesis, TU Delft, May 2009.

[7] Linux Foundation, *Iproute2*, http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2, November 2009.

[8] ______, *Netlink*, http://www.linuxfoundation.org/collaborate/workgroups/networking/netlink, November 2009.

[9] Holger Frning, *the htx board*, http://ra.ziti.uni-heidelberg.de/index.php?page=projects&id=htx.

[10] Thomas Graf, *nl_fib_lookup.c*, http://libnl.sourcearchive.com/documentation/1.1-5/nl-fib-lookup_8c-source.html.

[11] 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.

[12] S.H. Katamaneni, *Longest prefix match and incremental updates for range tries*, Master's thesis, TU Delft, August 2010.

[13] Michail Litvak, *Manpage ip*, http://linux.die.net/man/8/ip, May 2011.

[14] Matthew G. Marsh, *Iproute2 utility suite howto*, http://www.policyrouting.org/iproute2.doc.html, May 2011.

[15] Stefan Nilsson and Gunnar Karlsson, *Ip-address lookup using lc-tries*, 1998.

[16] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous., *Survey and taxonomy of ip address lookup algorithms*, March/April 2001, pp. 8– 23.

[17] David Slogsnat, Alexander Giese, Mondrian Nüssle, and Ulrich Brüning, *An open-source hypertransport core*, ACM Trans. Reconfigurable Technol. Syst. **1** (2008), 14:1– 14:21.

[18] I. Sourdis, R. de Smet, G. Stefanakis, and G. N. Gaydadjiev, *Data structure, method and system for address lookup*, October 2010.

[19] I. Sourdis, G. Stefanakis, R. de Smet, and G. N. Gaydadjiev, *Range tries for scalable address lookup*, October 2009, pp. 143– 152.

[20] Ioannis Sourdis, Ruben de Smet, and Georgi N. Gaydadjiev, *Range trees with variable length comparisons*, Proceedings of the 15th international conference on High Performance Switching and Routing (Piscataway, NJ, USA), HPSR, IEEE Press, 2009, pp. 42– 47.

[21] 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.

[22] G. Stefanakis, *Design and implementation of a range trie for address lookup*, Master's thesis, TU Delft, July 2009.

[23] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G. Kuzmanov, and E. Moscu Panainte, *The molen polymorphic processor*, IEEE Transactions on Computers (2004), 1363– 1375.

[24] Priyank Warkhede, Subhash Suri, and George Varghese, *Multiway range trees: scalable ip lookup with fast updates*, Computer Networks (2004), 289 303.

Linux Routing Table using IPROUTE2

A

The following parts come from the manpage of IP(8) [13]. These function calls are used to define and delete extra IP addresses for a given device. These will be used as endpoints for the added routes. To easily clear all added routes, they will be stored in a separate routing table, so this one can be flushed without problems afterwards. Also the functionality to do a routing table lookup is given. For more information about the IPROUTE2 suite and its usages please go to [14]

ip [*OPTIONS*] *OBJECT* {*COMMAND*|**help**}

OBJECT := { **link** | **addr** | **route** | **rule** | **neigh** | **tunnel** | **maddr** | **mroute** | **monitor** }

OPTIONS := { **-V**[*ersion*] | **-s**[*tatistics*] | **-r**[*esolve*] | **-f**[*amily*] { **inet** | **inet6** | **ipx** | **dnet** | **link** } | **-o**[*neline*] }
For adding or deleting ipaddresses to certain devices.

ip addr { **add** | **del** } *IFADDR* **dev** *STRING*
For showing and flushing the ip-addresses matching all of the given filters

ip addr { **show** | **flush** } [**dev** *STRING*] [**scope** *SCOPE-ID*] [**to** *PREFIX*] [*FLAG-LIST*] [**label** *PATTERN*]
How to list or flush the routes in a certain *SELECTOR*, for example TABLE 1.

ip route { **list** | **flush** } *SELECTOR*
The method to perform a route lookup for a given address, possibly with a from address and input or output interfaces.

ip route get *ADDRESS* [**from** *ADDRESS* **iif** *STRING*] [**oif** *STRING*] [**tos** *TOS*]
To perform addition, deletions and changes of single routes. The *ROUTE* may also contain the TABLE 1 part to point to a specific routing table for the given route.

ip route { **add** | **del** | **change** | **append** | **replace** | **monitor** } *ROUTE*

B Executing route lookups using LIBNL

```
/*
 * src/nl-fib-lookup.c         FIB Route Lookup
 *
 *     This library is free software; you can redistribute it
    and/or
 *     modify it under the terms of the GNU Lesser General
    Public
 *     License as published by the Free Software Foundation
    version 2.1
 *     of the License.
 *
 * Copyright (c) 2003-2006 Thomas Graf <tgraf@suug.ch>
 */


#include "utils.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getopt.h>
#include <stdarg.h>
#include <netlink/netlink.h>
#include <netlink/genl/genl.h>
#include <netlink/genl/ctrl.h>


static void print_usage(void)
{
        printf(
        "Usage: nl-fib-lookup [options] <addr>\n"
        "Options:\n"
        "    -t, --table <table>        Table id\n"
        "    -f, --fwmark <int>         Firewall mark\n"
        "    -s, --scope <scope>        Routing scope\n"
        "    -T, --tos <int>            Type of Service\n");
        exit(1);
}
```

```
int main(int argc, char *argv[])
{
        struct nl_handle *nlh;
        struct nl_cache *result;
        struct flnl_request *request;
        struct nl_addr *addr;
        struct nl_dump_params params = {
                .dp_fd = stdout,
                .dp_type = NL_DUMP_FULL,
        };
//      struct nl_dump_params params;
        int table = RT_TABLE_UNSPEC, scope = RT_SCOPE_UNIVERSE;
        int tos = 0, err = 1;
        uint64_t fwmark = 0;

        if (nltool_init(argc, argv) < 0)
                return -1;

        while (1) {
                static struct option long_opts[] = {
                        {"table", 1, 0, 't'},
                        {"fwmark", 1, 0, 'f'},
                        {"scope", 1, 0, 's'},
                        {"tos", 1, 0, 'T'},
                        {"help", 0, 0, 'h'},
                        {0, 0, 0, 0}
                };
                int c, idx = 0;

                c = getopt_long(argc, argv, "t:f:s:T:h", long_opts,
                    &idx);
                if (c == -1)
                        break;

                switch (c) {
                case 't':
                        table = strtoul(optarg, NULL, 0);
                        break;
                case 'f':
                        fwmark = strtoul(optarg, NULL, 0);
                        break;
                case 's':
                        scope = strtoul(optarg, NULL, 0);
                        break;
        case 'T':
              tos = strtoul(optarg, NULL, 0);
              break;
        default:
              print_usage();
        }
  }

  if (optind >= argc)
        print_usage();

  nlh = (struct nl_handle *) nltool_alloc_handle();
  if (!nlh)
        return -1;

  addr = (struct nl_addr *) nl_addr_parse(argv[optind],
     AF_INET);
  if (!addr) {
        fprintf(stderr, "Unable to parse address \"%s\": %s
           \n",
              argv[optind], nl_geterror());
        goto errout;
  }

  result = (struct nl_cache *) flnl_result_alloc_cache();
  if (!result)
        goto errout_addr;

  request = (struct flnl_request *) flnl_request_alloc();
  if (!request)
        goto errout_result;

  flnl_request_set_table(request, table);
  flnl_request_set_fwmark(request, fwmark);
  flnl_request_set_scope(request, scope);
  flnl_request_set_tos(request, tos);

  err = flnl_request_set_addr(request, addr);
  nl_addr_put(addr);
  if (err < 0)
        goto errout_put;

  if (nltool_connect(nlh, NETLINK_FIB_LOOKUP) < 0)
        goto errout_put;
        err = flnl_lookup(nlh, request, result);
        if (err < 0) {
                fprintf(stderr, "Unable to lookup: %s\n",
                    nl_geterror());
                goto errout_put;
        }

        nl_cache_dump(result, &params);

        err = 0;
errout_put:
        nl_object_put(OBJ_CAST(request));
errout_result:
        nl_cache_free(result);
errout_addr:
        nl_addr_put(addr);
errout:
        nl_close(nlh);
        nl_handle_destroy(nlh);
        return err;
}
```

C Shell files for doing the measurements

For doing the measurements there are 3 shell files KernelLookups.sh, HWlookups.sh and HWlookups2.sh. which are called as **For i in 1..1000; do ./*shellfile* $i >> *outputfile*; echo $i; done** This way the output (time required and amount of lookups) is written to *outputfile* while the user still can see the progress due to the echo $i. The KernelLookups.sh runs the getroute5 program to get the correct number of lookups from the kernel. HWlookups.sh takes care of the batchwise lookups and HWlookups2.sh handles the individual lookups to the hardware.

C.1 Measuring the kernel lookups

```
if [[ $# -gt 2 ]]
then
        nrLookups=$1
        nrRoutes=$2
        nrAddress=$2
        nrAdds=$3
        nrRemoves=$3
elif [[ $# -gt 1 ]]
then
        nrLookups=$1
        nrRoutes=$2
        nrAddress=$2
        nrAdds=$2
        nrRemoves=$2
elif [[ $# -gt 0 ]]
then
        nrLookups=$1
        nrRoutes=$1
        nrAddress=$1
        nrAdds=$1
        nrRemoves=$1
else
        echo "$0 [[destinations] routes] lookups"
        echo "When no destinations, a max of 255 will be added"
        echo "When no routes, a max of 255 will be added"
fi
if [[ $nrAdds -gt 255 ]]
```

```
then
        nrAdds=255
        nrRemoves=255
fi
if [[ $nrRoutes -gt 255 ]]
then
        nrRoutes=255
        nrAddress=255
fi

#echo "Going to add $nrAdds ip-addresses"
while [[ $nrAdds -gt 0 ]]
do
        /sbin/ip addr add 127.0.1.$nrAdds dev lo
        nrAdds=`expr $nrAdds - 1`
done

#echo "Adding $nrRoutes routes to routing table"
while [[ $nrRoutes -gt 0 ]]
do
        /sbin/ip route add 127.5.0.$nrRoutes via 127.0.1.`expr
           $nrRoutes % $nrRemoves` table 1
        nrRoutes=`expr $nrRoutes - 1`
done

#echo "Going to do $nrLookups lookups"
startTime=`date +%S%N`
while [[ $nrLookups -gt 0 ]]
do
        ./getroute5 127.5.0.`expr $nrLookups % $nrAddress` -t 1
            > /dev/null
        nrLookups=`expr $nrLookups - 1`
done
endTime=`date +%S%N`
totalTime=`expr $endTime - $startTime`
echo "$totalTime        us"

#echo "Flushing temporary routing table"
        /sbin/ip route flush table 1

#echo "Going to remove $nrRemoves ip-addresses"
while [[ $nrRemoves -gt 0 ]]
do
        /sbin/ip addr del 127.0.1.$nrRemoves/32 dev lo
        nrRemoves=`expr $nrRemoves - 1`
```

```
done
```

C.2 Measuring the hardware batch lookups

```
if [[ $# -gt 1 ]]
then
        First=$1
        Second=$2
elif [[ $# -gt 0 ]]
then
        First=$1
        Second=$1
else
        echo "$0 batchlookups [single lookups]"
        echo "When no destinations, a max of 255 will be added"
        echo "When no routes, a max of 255 will be added"
fi
startTime=`date +%S%N`
./copy2 $First 1 $First 1 > /dev/null
endTime=`date +%S%N`
totalTime=`expr $endTime - $startTime`
echo "$totalTime        us"

nrLookups=$Second
startTime=`date +%S%N`
while [[ $nrLookups -gt 0 ]]
do
        ./copy2 1 1 1 nrLookups > /dev/null
        nrLookups=`expr $nrLookups - 1`
done
endTime=`date +%S%N`
totalTime=`expr $endTime - $startTime`
echo "          $totalTime       us"
```

C.3 Measuring the hardware individual lookups

```
if [[ $# -gt 1 ]]
then
       Second=$2
elif [[ $# -gt 0 ]]
then
       Second=$1
else
       echo "$0 single lookups"
      echo "When no destinations, a max of 255 new routes will
         be added"
```

```
    echo "When no routes, a max of 255 routes will be added"
fi

nrLookups=$Second
totalTime=0
while [[ $nrLookups -gt 0 ]]
do
        startTime=`date +%S%N`
        ./copy2 1 1 1 nrLookups > /dev/null
        endTime=`date +%S%N`
        nrLookups=`expr $nrLookups - 1`
        intermTime=`expr $endTime - $startTime`
        totalTime=`expr $totalTime + $intermTime`
done
echo "$Second    $totalTime        us"
```

D Measurements

Timings (in nanoseconds) for the for different measurements

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
1	5005374	3836941	30213096	3875313	4101868	3996310
10	35914349	3939206	57908197	3721762	4068866	4259099
25	89187688	4120937	140530709	3708935	4084424	4088397
50	173711314	4424383	279769225	3931511	4117611	4278403
75	265578624	4662750	416368602	3817711	3999630	4018451
100	353333395	4908068	562706648	3784999	4093989	4213617
150	529112619	5609523	856400603	3951495	4213895	4095846
200	722158919	6220822	1130806999	3918796	4074584	4222996
250	893010046	6723897	1423163301	4326578	4163291	4225430
300	1060355173	7180500	1725196423	4293121	4019943	4120565
350	1227472994	7693216	1792333832	4393795	4211821	4102229
400	1405097726	8035925	2075024953	4538614	4161454	4088787
450	1572038989	8672929	2341968002	4548300	4189447	4146507
500	1733788375	9161732	2612332797	4805379	4122855	4109414
600	2080046453	10045556	3184360073	4951742	4215441	4185222
700	2452387189	11460622	3754006012	5045169	4215944	4077225
800	2773636677	12144149	4333220256	5399703	5852030	4090036
900	3132015593	13215609	4918009844	5503497	4285867	4253159
1000	3447066964	13933735	5614767183	5788067	4076653	4241729
2000		26674217		5885785		
3000		40075443		6518929		
4000		52340152		7466697		
5000		61381301		8370713		
6000		73945396		9332193		
7000		85997736		10335669		
8000		95602707		11192707		
9000		110347307		12223308		
10000		125136100		13051631		
11000		135846204		14053930		
12000		137277614		15232779		
13000		153517477		15914837		
14000		160316285		16662409		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
15000		190241213		17588437		
16000		189587806		19094041		
17000		209956795		19424995		
18000		216812524		20714836		
19000		227408586		21672653		
20000		253154860		22641088		
21000		253467553		23537365		
22000		280639977		24631848		
23000		284874458		25026751		
24000		289218353		26023062		
25000		303832694		26897582		
26000		307970718		25575729		
27000		333443753		28787346		
28000		328243365		26995585		
29000		342340857		30637088		
30000		378220690		27837380		
31000		367627741		29422832		
32000		395262901		30170864		
33000		401371335		36703327		
34000		412029354		28791232		
35000		430140057		30570415		
36000		450395038		34088980		
37000		419185556		38316884		
38000		448593258		33532724		
39000		488415628		40108253		
40000		461727114		34324508		
41000		490473549		44074690		
42000		494033192		42925072		
43000		533409266		43839449		
44000		533068432		42124146		
45000		537788577		35108204		
46000		526544208		35295667		
47000		595892704		44237721		
48000		554627199		35441825		
49000		604954539		45058467		
50000		546559292		45015885		
51000		542709725		45184953		
52000		565035385		37372040		
53000		568601664		38457593		
54000		583468717		42242144		
55000		584030536		36391827		
56000		598077276		46681559		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
57000		608656329		43614630		
58000		622679863		45997445		
59000		617599583		38175042		
60000		591704829		48340470		
61000		627869089		48824922		
62000		674210053		37613116		
63000		647396231		48842739		
64000		652835561		39887018		
65000		656531014		40790393		
66000		697225828		39794554		
67000		685841091		38125768		
68000		690350133		40422981		
69000		718730359		1040477346		
70000		692699944		40024885		
71000		762575347		40783438		
72000		749150429		40772956		
73000		774608390		39632032		
74000		795013090		39917558		
75000		760303960		45430076		
76000		803977541		42771694		
77000		794584077		41768278		
78000		778561534		46114941		
79000		835906306		42224505		
80000		815247202		43077263		
81000		819353663		43123610		
82000		837395351		53805101		
83000		882438360		44534380		
84000		871422624		46390806		
85000		893183489		44194346		
86000		931241836		44953712		
87000		890075058		43043665		
88000		886601117		48187491		
89000		930469750		44145140		
90000		979663232		43081494		
91000		927187536		45076470		
92000		938542729		46109653		
93000		967983176		46674628		
94000		973514765		48130690		
95000		1026624015		44275825		
96000		951574521		50257448		
97000		973903433		45257567		
98000		997931839		46798833		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
99000		1024064271		47740486		
100000		1029410407		46285928		
101000		1015311108		48526021		
102000		1059822989		48635540		
103000		1064932632		50267331		
104000		1073572648		51213178		
105000		1101623001		49514088		
106000		1108523267		50798151		
107000		1078999770		49672944		
108000		1091436636		52023317		
109000		1126387224		52688528		
110000		1169046418		49959168		
111000		1167594232		49714171		
112000		1159144384		50280268		
113000		1222194823		49509233		
114000		1167846239		50843324		
115000		1178373826		50886320		
116000		1251170295		49395460		
117000		1185666051		51654920		
118000		1243148384		53365884		
119000		1266883257		50726336		
120000		1261122749		55320162		
121000		1276302881		55591510		
122000		1246946462		55167672		
123000		1312577648		51161647		
124000		1391531197		53289493		
125000		1368256818		53562150		
126000		1314743020		55002037		
127000		1352237096		52208077		
128000		1368916717		53909696		
129000		1374008805		56005487		
130000		1390989855		56798377		
131000		1352222910		55801807		
132000		1376518802		53971825		
133000		1355445051		56484622		
134000		1331368213		54478290		
135000		1444831661		58926943		
136000		1479887417		55322141		
137000		1400266302		56568776		
138000		1464276198		57486876		
139000		1486170766		57516995		
140000		1431455831		56863786		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
141000		1561920483		62781550		
142000		1475770278		63935002		
143000		1501254162		58152706		
144000		1561548378		59797997		
145000		1548398861		68176940		
146000		1559473547		69484280		
147000		1595607719		60640435		
148000		1535155932		57164660		
149000		1620463786		71765379		
150000		1546976825		60943620		
151000		1589624679		74546059		
152000		1569204875		58324614		
153000		1536532853		71131242		
154000		1585925696		63929750		
155000		1641108741		62116493		
156000		1675746680		74596162		
157000		1687953831		73306377		
158000		1647648069		60075163707		
159000		1580666491		70921417		
160000		1649277771		72755874		
161000		1639149878		78222653		
162000		1642142923		73353694		
163000		1630681445		74516478		
164000		1648410272		70431829		
165000		1701066385		72313910		
166000		1705096846		76475038		
167000		1743144909		83134352		
168000		1701717695		83541697		
169000		1764023392		80124439		
170000		1747762988		83710804		
171000		1841335728		74144609		
172000		1744281976		80813227		
173000		1898803268		75313956		
174000		1825879290		81277720		
175000		1836520266		84858822		
176000		1860065848		85130759		
177000		1883814042		74225201		
178000		1915856788		85549289		
179000		1980731546		87716208		
180000		1887580489		86054066		
181000		1835495830		86490351		
182000		1904979092		86473596		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
183000		1992399270		82946037		
184000		1901702441		84994171		
185000		1952140499		87376092		
186000		1994628255		87302647		
187000		1978107187		84147537		
188000		1987819307		77194519		
189000		1977465049		84965929		
190000		2073065160		86772014		
191000		1935731796		85023556		
192000		2030857279		85662460		
193000		1984596597		88645115		
194000		1971613295		89834760		
195000		1971916663		85889740		
196000		2046611235		86083622		
197000		1926396682		89825273		
198000		1974555988		89139252		
199000		2006701384		1086866079		
200000		2040390463		90470126		
201000		2083942629		87289308		
202000		2161678650		80673036		
203000		2219402160		90839422		
204000		2090453201		1088031379		
205000		2048045393		88275850		
206000		2065153032		88774923		
207000		2121701642		88651101		
208000		2228460301		88689500		
209000		2129130143		89165605		
210000		2291567881		89803939		
211000		2384649763		89489548		
212000		2187077710		93895835		
213000		2152143716		89860262		
214000		2161557811		95841735		
215000		2337752655		94890245		
216000		2257497493		90761810		
217000		2243702413		90856350		
218000		2383582740		91136210		
219000		2272300046		93682074		
220000		2282306994		91587265		
221000		2324145350		97208450		
222000		2396272407		97099862		
223000		2249902644		92622087		
224000		2359601570		105085513		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
225000		2365070312		94723101		
226000		2390250984		99439232		
227000		2409246182		95339175		
228000		2339302342		100876397		
229000		2365857167		101138287		
230000		2374363832		101460093		
231000		2285425462		94143545		
232000		2350224687		107478465		
233000		2514903335		95744116		
234000		2377839945		102766885		
235000		2402238833		108537481		
236000		2498425743		104654013		
237000		2535012443		100402326		
238000		2555239484		109762634		
239000		2549259129		104917247		
240000		2385053948		110562636		
241000		2467372185		109444496		
242000		2462127437		99312916		
243000		2441039430		102613939		
244000		2546878739		110495186		
245000		2579761207		110994164		
246000		2479770345		109347645		
247000		2561892183		109569841		
248000		2620454268		111182366		
249000		2743458217		107288592		
250000		2551598726		106857227		
251000		2646558442		107995233		
252000		2525972733		105163611		
253000		2616988696		110222278		
254000		2724191559		99252678		
255000		2617098475		107989142		
256000		2626143565		107333349		
257000		2673395470		113472597		
258000		2766997470		109468196		
259000		2724597833		110276023		
260000		2652930325		111225358		
261000		2675439108		118551030		
262000		2865676774		111508376		
263000		2641478767		114806494		
264000		2833302808		114747496		
265000		2709169241		117455686		
266000		2693261833		114498784		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
267000		2735634013		115911783		
268000		2708413108		115648514		
269000		2937573007		113412621		
270000		2812914143		113307471		
271000		2921464198		114782037		
272000		2847120084		116651056		
273000		2814325401		117163393		
274000		3021226589		119036787		
275000		2923538417		117436472		
276000		2766247182		115488983		
277000		2782788741		117719204		
278000		3001480486		118578461		
279000		2897775124		118522899		
280000		2992030624		124513746		
281000		2887050905		132490912		
282000		2928640218		117382898		
283000		2902130310		129390873		
284000		3014575171		123161802		
285000		3148430736		123874492		
286000		3055714444		119605565		
287000		2878334971		126239178		
288000		2857035891		121862664		
289000		2915461447		122738019		
290000		3067072485		126188165		
291000		3036761451		128316256		
292000		2938827943		125095830		
293000		2971807223		122534231		
294000		3060675218		125634016		
295000		3132953301		127941083		
296000		2996327178		123810871		
297000		3174815454		124564404		
298000		3250567163		121349980		
299000		3012721676		124594495		
300000		3170306974		124987859		
301000		3005014025		128586299		
302000		3290731227		132309876		
303000		3142235682		126850948		
304000		3179174562		135987977		
305000		3207565902		137083355		
306000		3108039073		132049516		
307000		3205881839		134958947		
308000		3160823622		131167876		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
309000		3232813160		131451636		
310000		3090094737		134772481		
311000		3138105571		128967624		
312000		3345823301		129541042		
313000		3271287051		135590113		
314000		3308962953		136590954		
315000		3096583454		140224009		
316000		3323833441		134045387		
317000		3238776717		131087655		
318000		3360557017		136645188		
319000		3463707156		131832621		
320000		3372601926		137965656		
321000		3195563240		132659361		
322000		3326452310		133731871		
323000		3390828749		134370136		
324000		3365337576		138920296		
325000		3452319794		135266532		
326000		3469146949		141474217		
327000		3408167796		138325663		
328000		3507408297		140874928		
329000		3416742398		135913263		
330000		3573388333		135630320		
331000		3571667550		145007042		
332000		3601060362		142605407		
333000		3625017484		139013935		
334000		3475096772		143803800		
335000		3438220692		142152104		
336000		3450530549		147494092		
337000		3555484092		139957676		
338000		3555133242		146015374		
339000		3386661870		145559948		
340000		3437801683		147458613		
341000		3711320187		144813758		
342000		3522302457		146968031		
343000		3709610467		149090645		
344000		3524505424		145660665		
345000		3643781143		148030854		
346000		3533186395		144960254		
347000		3687322535		142915894		
348000		3639547510		146047000		
349000		3708384274		144017948		
350000		3633226662		149296203		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
351000		3595186314		148309143		
352000		3641641072		145674365		
353000		3674510460		150298132		
354000		3634982112		147516712		
355000		3652289386		151773933		
356000		3646963670		147109032		
357000		3835651966		149993468		
358000		3723042683		150703392		
359000		3611876150		153227578		
360000		3666075287		151285682		
361000		3770688609		152435155		
362000		3614269920		153359625		
363000		3753729002		146622684		
364000		3776306204		151823225		
365000		3739772382		156299060		
366000		3989217874		153802617		
367000		3873706958		149709801		
368000		3844812808		156328424		
369000		3841963719		150349283		
370000		3985432340		151411506		
371000		4100599145		151598451		
372000		3695606136		150036910		
373000		3988882215		151051913		
374000		3857916281		158246185		
375000		3843960499		159863563		
376000		3972269067		152454170		
377000		3758049273		163377902		
378000		4307110809		152573435		
379000		4109288932		161812165		
380000		4036964924		161575481		
381000		4183483739		157488705		
382000		3813137209		168983646		
383000		4009284713		165907158		
384000		4260832615		161061301		
385000		3996761807		163835921		
386000		3953021085		164308165		
387000		3936807465		157154030		
388000		4059472123		159191356		
389000		3963230366		162244008		
390000		4118142317		166546541		
391000		4025048671		153295858		
392000		4050958617		167100269		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
393000		4062733870		168023500		
394000		3969900990		155903507		
395000		4219619541		170641986		
396000		4126703766		167713488		
397000		4111618048		162936503		
398000		4335447786		157537719		
399000		4113256993		166338663		
400000		4213363001		158966429		
401000		4367457792		173778013		
402000		4343716714		168965812		
403000		4113565517		159997988		
404000		4107984074		170821320		
405000		4154560294		159337608		
406000		4157825141		167743898		
407000		4189902559		1156965791		
408000		4149973604		175318576		
409000		4341517651		169492949		
410000		4230303014		159291196		
411000		4246212139		171352559		
412000		4183263562		174576812		
413000		4365114491		169486520		
414000		4440349686		173303404		
415000		4345262319		173983449		
416000		4326536632		176817524		
417000		4251904635		167028649		
418000		4230190934		166507754		
419000		4474545295		164111276		
420000		4521020966		164847930		
421000		4514083710		168995008		
422000		4581340157		175574278		
423000		4742715924		164645513		
424000		4469233755		159986121		
425000		4462599519		167223215		
426000		4307196406		166060223		
427000		4355155178		159312548		
428000		4346011266		175094649		
429000		4635981572		170928401		
430000		4892784506		157431358		
431000		4434427420		164321966		
432000		4715732095		166913145		
433000		4545112865		165021720		
434000		4467197080		168201663		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
435000		4744827710		165434779		
436000		4623046526		182214461		
437000		4417379002		180995953		
438000		4486894866		176441658		
439000		4605270776		1184152767		
440000		4452448405		166074549		
441000		4483493027		170140450		
442000		4556598145		170177899		
443000		4499096206		169186210		
444000		4494077753		177399019		
445000		4502626982		184289072		
446000		4520598560		180674780		
447000		4672005315		170139273		
448000		4661289957		168031231		
449000		4667577834		177773681		
450000		4708782465		170283345		
451000		4994639087		177653231		
452000		4892217545		181315896		
453000		4984228988		180340073		
454000		4803453466		176908712		
455000		4997706764		169658234		
456000		4691140201		181926227		
457000		4877980652		171354686		
458000		4721058898		173565532		
459000		4652426697		183301371		
460000		4761372009		1169937170		
461000		5013866165		168842223		
462000		5040126640		182503225		
463000		4795483336		185741357		
464000		4686068824		179563826		
465000		5050411198		186696350		
466000		4767083225		170081419		
467000		4784135443		174980637		
468000		4892105946		180161996		
469000		4939525303		176083364		
470000		4923554048		187716122		
471000		4945306821		1181394378		
472000		5155167953		176168217		
473000		5066048561		188316338		
474000		4760149645		193868526		
475000		4922635855		188455498		
476000		5432958984		186166150		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
477000		5043464186		187914302		
478000		5160989513		180729840		
479000		5054309569		173426389		
480000		5153772753		164437900		
481000		4979164471		178430878		
482000		4885645218		194668322		
483000		5158063622		189055492		
484000		5295324260		178292815		
485000		4948156099		177323424		
486000		4916968753		175280649		
487000		4884607279		174536383		
488000		5355467383		183528172		
489000		5191900506		196293894		
490000		5281444564		177594553		
491000		5440113835		178690781		
492000		5326320884		177893134		
493000		5207045340		182870961		
494000		5161172044		181104101		
495000		5070209886		173412769		
496000		5182962455		185822820		
497000		5276546673		177373991		
498000		5461278195		186265218		
499000		5171637406		183760742		
500000		5067888298		184024542		
600000		6140249920		238612961		
700000		7209054607		259866925		
800000		8236118251		279157350		
900000		9434565653		304725873		
1000000		11235154757		330850076		
1100000		11161901462		349071798		
1200000		12383578279		385882299		
1300000		13391284873		408763658		
1400000		14260795360		428883720		
1500000		15294363631		460988305		
1600000		17671871603		475374514		
1700000		17456319777		500299153		
1800000		18112047827		525218755		
1900000		19506193586		546425440		
2000000		20166329879		566629566		
2100000		21488513571		586338588		
2200000		23114020028		598983508		
2300000		23848062366		638576760		

lookups	SW Single	SW Batch	HW Single	HW Batch	HW Batch 1 Updates	HW Batch 10 Updates
2400000		25225411925		654380621		
2500000		27499414705		678711464		
2600000		27498750300		696341276		
2700000		28473986037		713487483		
2800000		28722691312		745602631		
2900000		31550811717		768087103		
3000000		30469132949		791105421		
3100000		32526370644		810248211		
3200000		33542710377		836787485		
3300000		34929069277		855539321		
3400000		35183594766		884037504		
3500000		35460149303		904326121		
3600000		38984195247		925819680		
3700000		37771406163		949967684		
3800000		40243529303		971875478		
3900000		41014842020		992833088		
4000000		41224051901		1009484610		
4100000		43432022474		1037170067		
4200000		43695930040		1055718498		
4300000		45346339869		1078303171		
4400000		45575213909		1106614385		
4500000		46875866589		1129932354		
4600000		48040516903		1151569008		
4700000		49776731109		1169314690		
4800000		49134584061		1205313980		
4900000		50992185815		1219714515		
5000000		55249892855		1234610640		

Communication scheme between host and hardware implementation

E

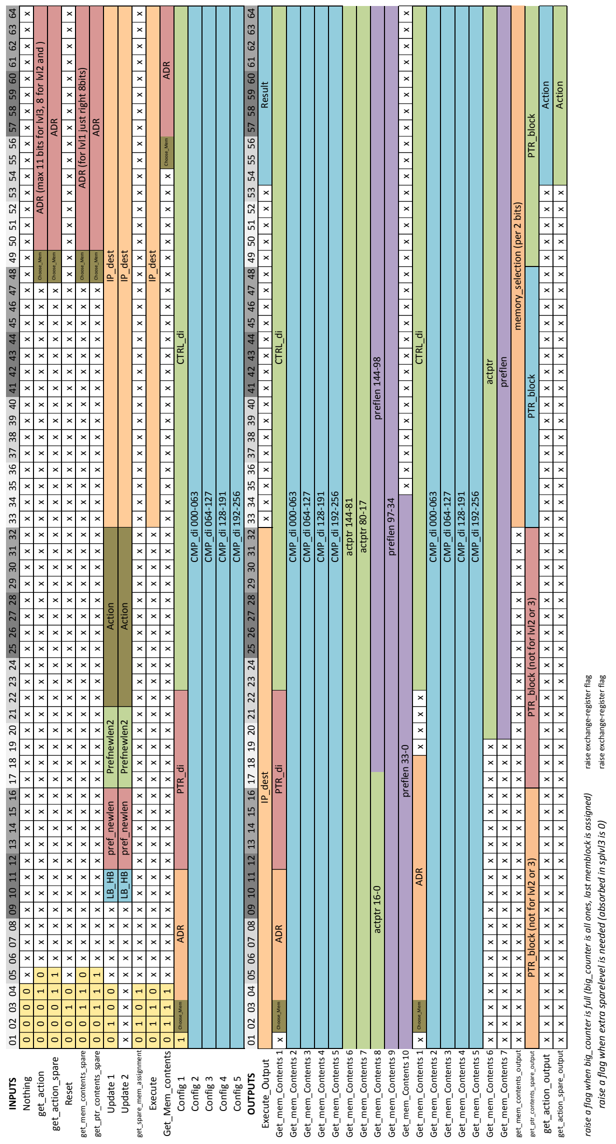


Figure E.1: *Communication scheme between the host and hardware Range Trie implementation.*

Curriculum Vitae

Dion (D.J.A.) van Adrichem, born in Rotterdam on January 21st of 1988. Has recevied his Bachelor of Science in Electrical Engineering at the Technical Univerity of Delft in 2009 with the project: *Design of an LED poi.* During the Bachelor project a LED poi is designed and prototyped. With the usb-programmable poi it is possible to display images and videos as well as plain text. While swinging the poi; LEDs will project the required content. Together with the accelerometer the poi knows its position and switches the LEDs on/off accordingly.

Currently Dion is working on his Masters thesis at the department of Computer Engineering. During the project a complete design of the Range Trie is implemented in an HTX reconfigurable system which supports hardware accelerated functions (the Range Trie). The hardware implementation of the Range Trie is compared to the Linux routing table to measure its performance and whether or not it might be an interesting field for further research.

Next to his work on his Masters thesis, Dion is also co-owner of ITCall. ITCall is a Dutch IT service-provider for call and contactcenters. With ITCall, Dion has gained much experience in the different callcenter applications like Vocalcom, Teleknowledge, Interactive Intelligence and ABC Solutions. Next to the different contactcenter applications including their specific scriptinglanguages, he has gained much expierence in databases including Integration Services and Reporting Services, networking, VoIP and quality ensurance.

Next to his study and work, Dion is a sports fanatic and not the kind that sits on the couch. Next to playing badminton, running and scuba-diving he also likes to sit back and enjoy the company of his friends.

Contact information:
Dion van Adrichem / dvanadrichem@hotmail.com / +31 6 1076 4545