ANCS (Symposium on Architecture for Networking and Communications Systems), October 2005, pp. 81-90

A Novel Reconfigurable Hardware Architecture for IP Address Lookup

Hamid Fadishei
Computer and IT Department
Amir-Kabir University of Technology
Tehran, Iran
fadishei@ce.aut.ac.ir

Morteza Saheb Zamani
Computer and IT Department
Amir-Kabir University of Technology
Tehran, Iran
szamani@ce.aut.ac.ir

Masoud Sabaei
Computer and IT Department
Amir-Kabir University of Technology
Tehran, Iran
sabaei@ce.aut.ac.ir

ABSTRACT

IP address lookup is one of the most challenging problems of Internet routers. In this paper, an IP lookup rate of 263 Mlps (Million lookups per second) is achieved using a novel architecture on reconfigurable hardware platform. A partial reconfiguration may be needed for a small fraction of route updates. Prefixes can be added or removed at a rate of 2 million updates per second, including this hardware reconfiguration overhead. A route update may fail due to the physical resource limitations. In this case, which is rare if the architecture is properly configured initially, a full reconfiguration is needed to allocate more resources to the lookup unit.

Categories and Subject Descriptors

C.2.6 [**Computer – Communication Networks**]: Internetworking – *routers*.
B.7.1 [**Integrated Circuits**]: Types and Design Styles – *algorithms implemented in hardware*.

General Terms

Algorithms, Design, Experimentation, Performance.

Keywords

IP Address Lookup, Longest Prefix Matching, Reconfigurable Hardware, Hashing, Field-Programmable Gate Array (FPGA), Application Specific Integrated Circuit (ASIC).

1. INTRODUCTION

Increasing number of Internet users and the advent of new multimedia networking applications have resulted in growth of traffic rates on backbone links. There are three key factors to be considered in designing IP networks to maintain a good service: links with large bandwidth, high speed data switching and high packet forwarding rates. Currently, available optical data transmission links and current data switching technology allow easy handling of the first two factors. Therefore, deployment of high performance routers to forward packets is the key to the success of the next generation IP routers [1].

Many tasks must be done in IP packet forwarding: decreasing time-to-live (TTL) of the packet, checking the packet checksum for errors, updating the checksum etc. However, the bottleneck of the packet forwarding operation is IP address lookup or determining the corresponding output interface of the router according to the packet destination address to send the packet. When an Internet router routes a packet, it looks for its 32-bit destination IP address to see if it matches the entries in the router. Routing entries in the router are in different lengths and therefore multiple matches may occur. The router must find the longest match and use the corresponding output interface to send the packet. This makes IP address lookup a challenging problem.

The rest of the paper is organized as follows. In Section 2, a brief description of the prior works related to IP address lookup problem is given. Two previous works, which use reconfigurable hardware, are described in Section 3, after an introduction to reconfigurable hardware. In Section 4, we present our proposed approach whose implementation is explained in Section 5. Section 6 shows the results and comparisons. Section 7 concludes the paper.

2. RELATED WORK

Many software and hardware IP address lookup approaches are proposed by researchers. A successful approach proposed for this problem must be:

- fast in performing IP lookup operations. Considering IP packets of 1000 bits long on average, a 100 Gbps router needs a lookup engine capable of working at a rate of 100 Mlps.
- fast in updating its routing database, i.e. adding or removing prefixes. It must anticipate a burst of 100 or more update operations per second [14].
- scalable in terms of routing tables growth caused by the increasing number of Internet users.
- scalable for migrating from IPv4 to IPv6.
- efficient in terms of implementation costs, achieving above objectives at a minimal cost.

Any IP address lookup approach can be scored according to the above parameters [14].

ANCS'05, October 26–28, 2005, Princeton, New Jersey, USA.

Binary trie which is the base of many existing schemes is a tree data structure that is searched starting from its root and going down to left or right depending on the value of the current bit of the address being searched. Any time a prefix is seen at a node of the trie, it is stored as the longest matching prefix up to now.

The basic trie structure needs up to *w* memory accesses per lookup, where *w* is the number of distinct prefix lengths existing in the routing database. The number of memory accesses to the trie is optimized using path compression [11] and level compression techniques [12]. The former uses multiple branches from every node, checking multiple bits of the destination address at each step. This results in better search performance at the cost of more memory usage and worse update time because some prefixes need to be expanded. The level compression technique, on the other hand, removes the nodes with only one child but the number of removed nodes must be stored somewhere for later searches. This technique is only useful for sparse tries but not for huge tries of backbone Internet routers in which there are not many one-child nodes.

Researchers implemented the trie in various ways both in hardware and software. Some of them used the straightforward node and pointer structures used in the software implementation of the tree structures [4] [12]. Some others implemented the trie as bit vectors [13] [18]. Trie was also implemented on reconfigurable hardware as a reconfigurable finite-state machine (FSM) [3] or a reconfigurable binary decision diagram (BDD) [15]. The following section describes these two implementations in details.

Modified versions of the binary search algorithm have been used in IP address lookup problem [8] [21]. The modifications were to make the standard binary search algorithm applicable to searching for the longest matching prefix. These approaches were extended to multi-way search. Both [8] and [21] had a good search time, but were poor in terms of the time required to update the routing table.

Content addressable memories (CAMs) were also used for IP address lookup [10]. A CAM is a memory that can be searched for a data item in parallel. There is a kind of CAM called ternary CAM (TCAM) which can be given a mask to be used for the desired length of data items in its search process. Therefore, if prefixes are sorted by their length, the first match will be the longest one. TCAMs are expensive and have a high cost of updating routes because all prefixes must be kept sorted by length.

Trie requires large memory because it uses the IP address itself as an index to the data structure. Hashing technique can also be used for IP lookup where a hash function of each item is computed and used as an index to a table to insert the item. To search for an item, if its hash function is computed, we only need to search the table row to which this hash value points.

Hashing is usually used for exact matching. For IP lookup, which must find the longest matching prefix, prefixes of each length are searched for exact matches with the IP address using hashing and then, the longest match among them is chosen. Waldvogel et al [19] used binary search to find the longest match among the hash results for every length. However, in their scheme, it was not necessary to use hashing for all lengths, but only some lengths were searched for an exact match during the binary search. To make this possible they added extra information to routing data. However, the computation of this information lengthens the update time. In [9], a hardware-based method uses hashing for all lengths in parallel and finds the longest match among the results using a priority encoder. The scheme in [19] is suitable for software implementations while the method in [9] takes advantage of parallelism in hardware. Our proposed scheme is an improved version of [9] using the benefits of hardware reconfiguration.

3. RECONFIGURABLE HARDWARE AND IP ADDRESS LOOKUP

Recently, reconfigurable hardware has been used for IP address lookup. Reconfigurable hardware is a kind of hardware whose functionality may change in response to the demands placed upon the system while it is running. This gives us both the flexibility of software and the performance of hardware.

This capability of hardware reconfiguration can be beneficial from different aspects. Some of these benefits are listed below:

- As much performance as possible can be gained from limited hardware resources by tuning configuration parameters on a flexible hardware architecture.
- Hardwired data structures can be used to accelerate performance by eliminating the drawbacks of traditional memory storage-based data structures.
- The system cost can be reduced by fitting multiple features and applications on a single reconfigurable hardware platform or by partitioning an application into some stages being configured serially on a smaller platform.

Some of currently available field-programmable gate arrays support partial reconfiguration in a few microseconds and full reconfiguration in a few milliseconds. This has made reconfigurable systems more attractive to researchers due to their capability of implementing reconfigurable hardware in real world.

References [3] and [15] used reconfigurable hardware to achieve high performance trie implementations. Both approaches profited from the second of the three advantages of reconfigurable hardware mentioned above. As seen in Figure 1, the lookup engine is a reconfigurable hardware containing the hardwired routing data. Therefore, the engine performs high speed lookup decisions with no memory accesses. However, any route update should affect the hardwired data by reconfiguring the engine. This task is done by the reconfiguration controller unit. It translates the routing table updates to hardware reconfiguration information and applies it to the reconfigurable platform. We use the first and the second advantages of reconfigurable hardware.

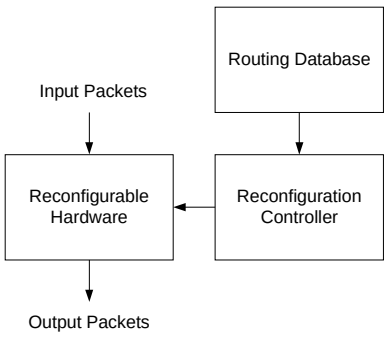


Figure 1. IP address lookup using reconfigurable hardware.

Desai et al [3] implemented the binary trie as a reconfigurable finite-state machine. They considered the binary trie as an FSM, the states of which represent the nodes of the trie and the links represent the transitions between the states. Then, that huge FSM was partitioned into smaller sub-FSMs to be implementable as an ASIC. These sub-FSMs were cooperated via a main controller. Each time the routing table needs an update, the trie must change and this change must be reflected to the FSM with a hardware reconfiguration. Sangireddy et al [15] used binary decision diagrams to implement the binary trie as a reconfigurable BDD. They partitioned the trie into five smaller tries according to the bits of the result output interface and simplified the corresponding BDDs and converted the resulted BDDs to logic implementable on lookup tables of a reconfigurable FPGA. Like Desai's scheme, any routing table update results in a reconfiguration that must be applied to the architecture using a reconfiguration controller.

The reconfigurable architecture in Figure 1 makes an obvious consideration necessary: incremental route updates which only need a partial hardware reconfiguration must be made possible; otherwise the huge rate of route updates can make the approach impractical on a backbone router.

Desai et al [3] hope each route update to affect only one of the many sub-FSMs but there is no guarantee for that. Sangireddy et al [15] say that the small BDD differences corresponding to two adjacent routing table snapshots result in small logic variations. This makes incremental route updates probable but as difficult as they do not offer any way to do that.

Another problem arises for large routing table sizes. Currently, a backbone router may have as many routing prefixes as 100K which needs a huge reconfigurable hardware platform for hardwired implementation of the FSM or BDD corresponding to its trie.

The proposed architecture in this paper can address both problems by taking advantages of both traditional hashing scheme and reconfigurable hardware. The problem of large hardwired data size is addressed by only implementing the collided entries on reconfigurable hardware. Other prefixes can be searched with a single memory access to the main table. We also addressed the problem of high route update cost by proposing a novel reconfigurable search tree architecture for collision resolution. The proposed architecture can be updated with partial reconfigurations when adding or removing prefixes.

4. OUR PROPOSED APPROACH

First, we take a closer look at the basic hashing scheme used by Lim et al [9]. Then we explain our improvements to reduce the collisions of hashing operation and finally, the novel reconfigurable search tree architecture to resolve the hash collisions is presented.

4.1 The Basic Hashing Scheme

Hashing is useful in exact matching problems. For a longest matching problem, such as IP address lookup, each existing length of routing prefixes is searched for a match using hashing and the longest one among the possible match results of each length is chosen. This selection can be done using a priority encoder in hardware. Figure 2 shows this method. Each engine searches one existing length of prefixes for an exact match and reports the result to the priority encoder. Figure 3 shows the structure of each engine in which used memory words are marked. The hash value, computed according to the first N bits of the address, is used as an index to the main table. When inserting a new entry, the hash value may point to a used word of the main table. For such a collision, the new value must be stored in a subtable. When searching in this structure, the row of the main table to which the hash of the address points is inspected. We are done if a match is found in the main table. Otherwise, search continues at the subtable of this row of the main table if one exists. Subtables of different rows may vary in size. A good hash function is the one which distributes data elements uniformly in the main table's address space which results in the subtables of almost similar sizes.

4.2 Multi-Column Main Table

Implementing parallel engines requires separate memories as their main table. In the current technology, it is possible to include multiple SRAM blocks on a chip easily. This on-chip memory distribution can be more beneficial to improve the hashing quality. A multi-column main table can be used in which each column is implemented as a separate memory. The columns in the main table are checked for a match in parallel in one memory access time by a simple circuit. This reduces subtable sizes and moves many collisions from subtables to the main table. Reducing the size of subtables is important in our proposed scheme because they are implemented in reconfigurable hardware. As mentioned earlier, implementing the whole routing table as hardwired data, as in [3] and [15], is hardly practical on a reconfigurable hardware platform due to its big size. This also results in worse average route update time because all update operations need a hardware reconfiguration whose overhead is considerable on current reconfigurable hardware platforms. Our proposed approach addresses these two problems by holding the majority of routing prefixes in the multi-column main tables. A two-column main table is shown in Figure 4(a). It can be seen that the subtables are shrunken but at the cost of using more memory space for the same number of prefixes. Increasing the number of columns reduces subtable sizes more. On the other hand, more memory space may be wasted if the number of columns is increased (Figure 4(b)). This increased memory usage can be justified for two reasons:

- If a good hash function is chosen to uniformly distribute prefixes in the address space of the main table, a reasonable number of columns can be decided for the main table so that the main table is almost full and very few subtables are needed.
- The multi-column main table still has less memory requirements than many other implementations as the hashing scheme well reduces the memory usage.

4.3 Reconfigurable Search Tree

Hash subtables are usually searched using binary search algorithm. Conventional binary search tree is suitable for software implementations but in hardware, memory allocation is a great concern. Although it is not impossible, it is not easy, specially if more complicated binary search trees such as AVL trees are used to hold the tree balanced after update operations.

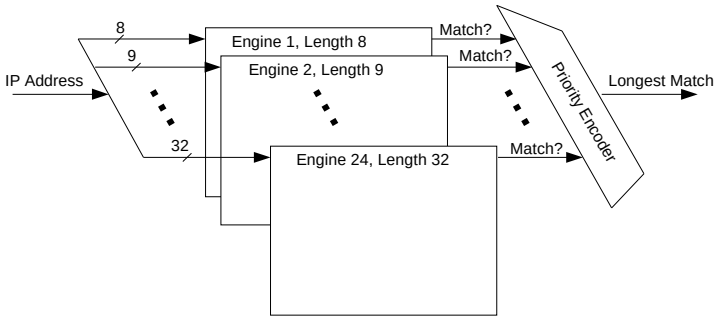


Figure 2. Using hashing for IP address lookup.

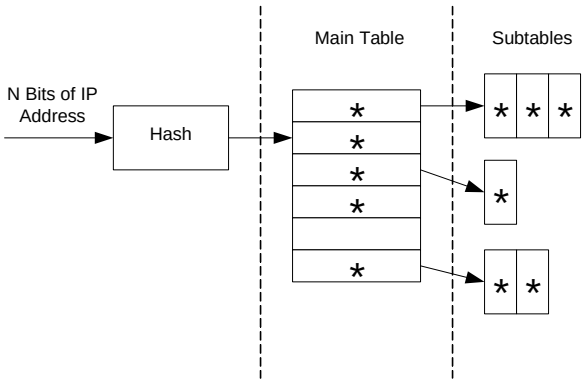
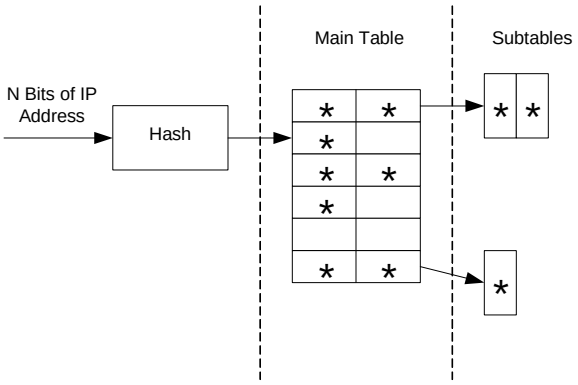
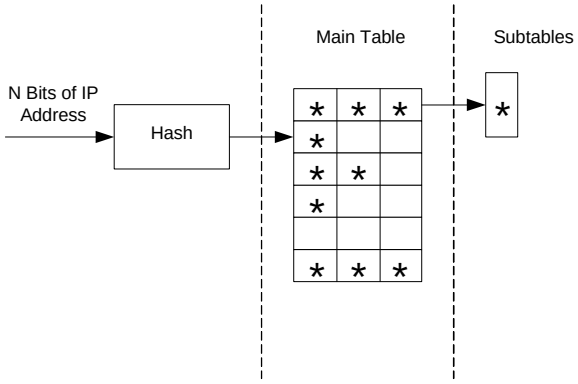


Figure 3. Structure of each engine of Figure 2.



(a)



(b)

Figure 4. Multi-column main table; (a) 2 columns (b) 3 columns.

We propose a novel reconfigurable search tree architecture the nodes of which are the reconfigurable hardware logic blocks. This is a fixed skeleton complete tree in which hardwired data elements are stored in the reconfigurable hardware resources of each node. Unlike the conventional binary search tree, a new prefix can be inserted in any node of the tree, making partial reconfiguration possible for update operations. Each node has a simple logic; it stores a prefix and the next hop information related to this prefix. Data flow direction in the tree is from child to the root. Each node compares the prefix to be searched with its hardwired prefix. The comparison result is reported to the parent node. If a match is found in the child node, the next hop information will be given to the parent node; otherwise, any node which gets the next hop information from one of its children only passes it to its parent. The final match result is obtained from the root node. Figure 5 shows the structure of one node. Reconfigurable areas are shown as shaded blocks. The node operates as follows.

If the node gets a *found_left* signal from the left child, it transfers the *nhp_left* value to its parent. Otherwise, if a *found_right* signal is seen, the *nhp_right* value is transferred to the parent. If no found signal is reported from neither the left nor the right child and if the node is configured to have a valid data (configured with *Valid* bit equal to 1), it compares the *prefix_search* with its hardwired prefix value. If they match, the *NHP* value is sent to the parent via *nhp_out* port. *found_out* signal goes up every time the node wants to inform its parent that a valid *nhp_out* value is being sent to it.

Figure 6 shows how the nodes are connected together to build the tree. As seen in the figure, the simple structure of the tree makes pipelining easy. Pipe registers (dashed blocks in Figure 6) can be inserted between suitable levels of the tree. Anytime a prefix is entered in the pipeline, all leaf nodes get a '0' as the initial value of *found_left* and *found_right* signals. After traversing the pipe depth, prefix and the possible match result exit from the other side simultaneously. Some nodes of the tree are empty and ready to accept new inserted prefixes. These Passive Nodes (PNs) whose internal *Valid* bit is configured to be zero, do not decide on data stream and only pass it to their parent. Active Nodes (ANs) are those having a valid (*Prefix*/*NHP*) pair and compare the prefix being searched with their internal prefix value.

The amount of reconfigurable logic resources consumed by each node depends on the size of *Prefix* and *NHP* fields. For example, a tree node of length 24 utilizes 14 Virtex-II Pro FPGA slices in our case while 7 slices are enough for each tree node of length 12.

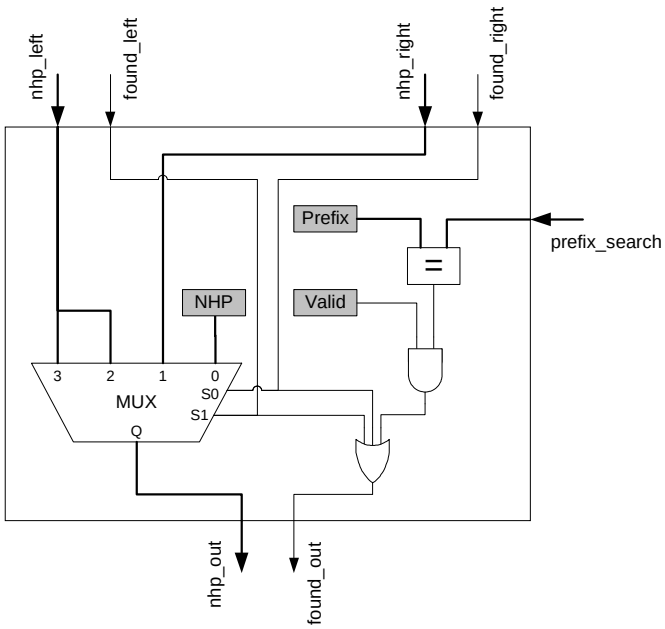


Figure 5. Structure of a node in the reconfigurable search tree.

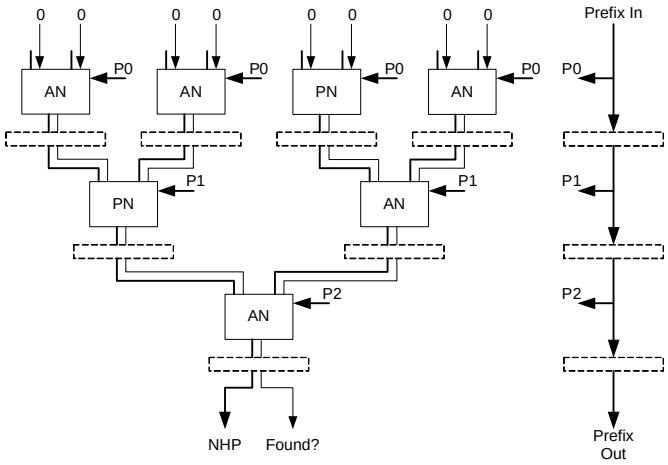


Figure 6. Building a reconfigurable search tree using the node in Figure 5.

4.4 Update Scenario

Adding and removing prefixes to/from the reconfigurable search tree has a bounded time of partially reconfiguring only some portions of a node. When a prefix is to be removed from the tree, only the *Valid* bit of the node containing the prefix is reconfigured to zero. For adding a new prefix to the tree, any empty node (any node with *Valid* bit equal to '0') of the tree can be selected for the insertion operation. The *Prefix* and *NHP* fields of this node are reconfigured to have the proper values and *Valid* bit is reconfigured to '1'.

Prefixes of all subtables of each length are integrated in a reconfigurable search tree. This tree is searched in parallel with the access to the main table. The reconfiguration controller introduced in Figure 1 has the responsibility of tracking the used and free nodes of the tree of each length. When an update operation fails in the main hash table, the reconfiguration controller is requested to continue the update operation in the reconfigurable search tree. In fact, only a small fraction of route updates are expected to need a reconfiguration because the majority of prefixes settle on the main hash table and most of update operations are done in the main table.

The reconfigurable search tree has a limited number of nodes. When all nodes of the tree are in use, new insertion requests are rejected. In such a situation, which is due to physical limitations, more nodes must be allocated to the tree using a full hardware reconfiguration. We will show that a proper number of nodes can be decided for the tree of each engine so that the tree can stably work for a long period of time without overflowing.

5. IMPLEMENTATION

The proposed architecture was synthesized on a Xilinx FPGA from Virtex-II Pro family. These series of FPGAs support fast partial reconfiguration and are well suited to our application.

5.1 The Overall Structure

Figure 7 shows the high level block diagram of a sample engine with a main table with 4 columns on the left hand side and a reconfigurable search tree with 7 nodes on the right hand side. Dashed blocks in the figure are pipeline registers. We used cyclic redundancy check (CRC) for the *Hash* block which was shown to be a very good hash function [6]. To reach a better performance, the hash function was implemented byte-wise and pipelined to be able to compute one hash value per clock cycle [16].

There is a *Match* block for each memory bank. It compares the prefix being searched with the value in the memory bank row to which the hash of the address points. If the memory contains valid data and if it matches the prefix, the corresponding *Select* node input is triggered.

Select nodes implement a simple logic: if they find a valid data in one of their two inputs, they pass the data; otherwise they hold their output at invalid state. Using this property, we can build a pipelined selection tree to find the possible match in the main table with N columns with a latency of $log_2(N)$ and a throughput of one operation per clock cycle. In our implementation, we put pipeline registers between every two levels of the selection tree because *Select* nodes have a short delay. This halves the tree latency without affecting the pipeline clock. *Select* nodes have another useful property: Their left input has a higher priority than the right one when both inputs have valid data. This implies an overall priority from left to right in the inputs of the selection tree. Therefore, it can also be used as the priority encoder in Figure 2 to find the longest match among the results of all engines.

Search process advances in the main table in parallel with the reconfigurable search tree. The number of pipe stages of both parts must be equalized using additional pipe stages for the shorter one. For example, two extra pipe stages are added to the reconfigurable search tree in Figure 7.

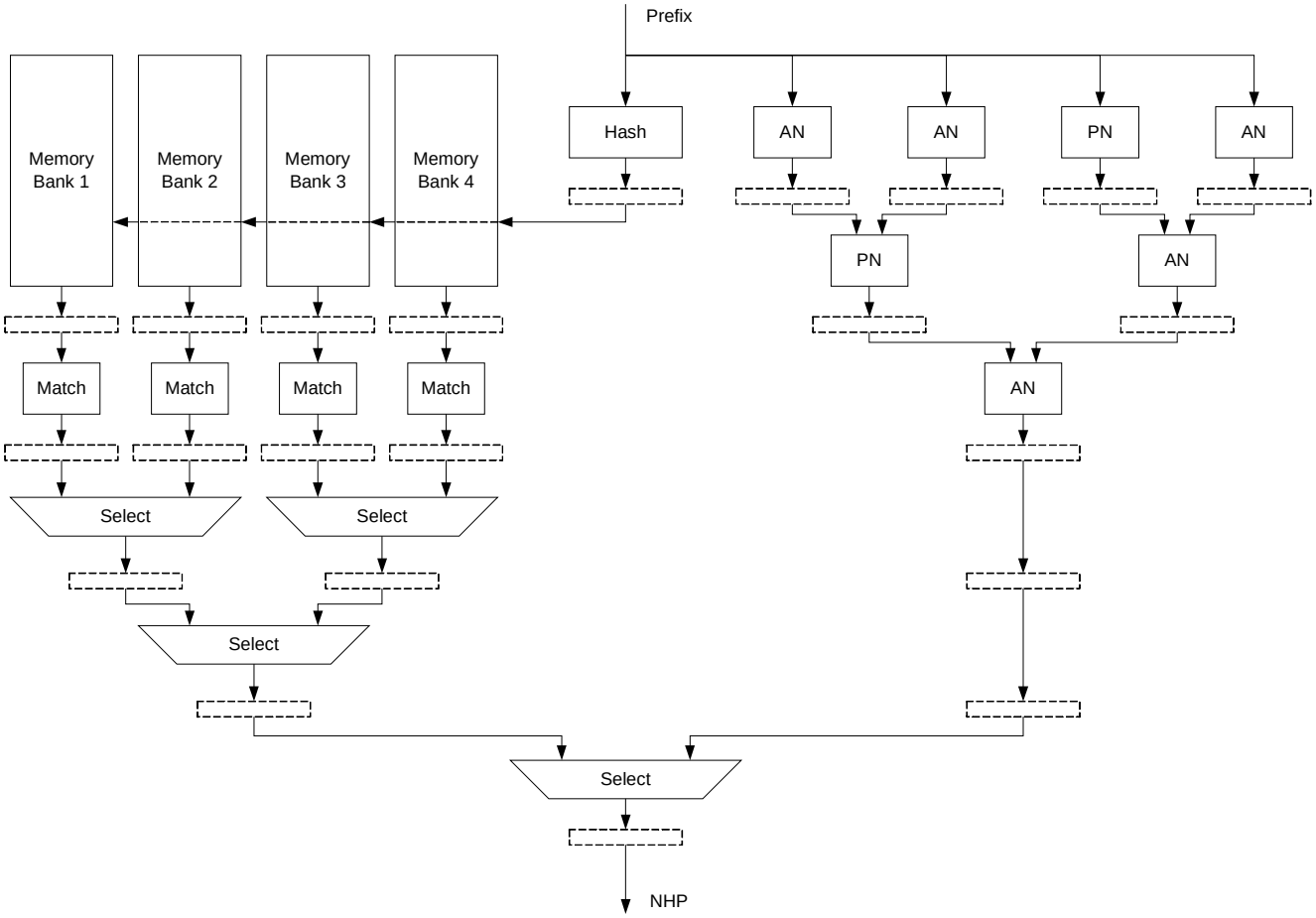


Figure 7. Structure of a sample engine used for each particular length of prefixes.

The possible match results of the main table and the reconfigurable search tree go into a *Select* node to generate the output result of the engine. The results of all engines must go into a selection tree which finds the longest match. Again, additional pipe stages may be needed between the output of some engines and the input of the selection tree to make the latency of all engines equal.

The powerful language constructs of VHDL allowed us to write a flexible and synthesizable code. Pipeline stage balancing and tree node instantiation of the reconfigurable search trees and the selection trees were handled automatically. This was done after determining the number of reconfigurable search tree nodes and the size and the number of main table memory banks for each engine via some VHDL *generic* parameters. This is important as it can help to perform automated full reconfigurations when a reconfigurable search tree overflows.

5.2 Flexible Architecture Description

We proposed a flexible architecture with some initial configuration parameters. Initially, it is not necessary to implement engines of all lengths. Our routing table may lack some lengths which allows us to remove these engines from the design.

Another parameter is the size of the hash main table. If you compare Figures 3, 4(a) and 4(b) you will see that the number of columns and rows of the main table affects total memory usage and subtable sizes. As said before, a reasonable size of the main table must be decided to trade-off between the memory wasted in the main table and the size of subtables. After determining the space to be allocated to the main table, the number of rows and columns which make that space are to be decided. The granularity of distributed memory elements on the target chip is effective to this decision.

In addition, the number of nodes for each engine in its reconfigurable search tree must be determined. The number of nodes must be greater than the number of collided prefixes to allow further prefix additions to the tree.

These decisions are dependent on the number of prefixes and routing data distribution which varies in different routers. Therefore, having a flexible architecture which permits tuning of the above parameters is useful. Parameters can be tuned easily by setting some VHDL *generic* parameters according to the routing table characteristics of a particular Internet router once. Then, the architecture is synthesized on a suitable reconfigurable hardware platform.

5.3 Automated Generation of Parameters

To make the analysis of the proposed architecture simple, we defined three basic parameters described as follows. The first parameter describes the ratio of memory space allocated to the main table of any length to the existing prefixes of this length. This parameter is referred as *MPR*. Using this parameter, the total size of the main table will be determined for a particular routing table. The number of rows and columns which make that space are determined according to the second parameter called *WHR*. This parameter defines the width to height ratio of the main table. After determining the size of the main table, prefixes can be inserted into the main table and the number of collided entries is determined. To allow further prefix additions, we allocate more nodes than the collided entries for the reconfigurable search tree. This defines the third parameter as *RCR* which is the ratio of the number of node in the reconfigurable search tree to the number of collided entries.

These parameters can be used after tuning to automatically adjust the flexible architecture for a particular routing table.

6. EXPERIMENTAL RESULTS

We used a snapshot of *rrc08* routing database [22] taken in 12:00 PM, March 1 2004 to tune the three basic parameters for this particular router. Figure 8 shows the distribution of 89979 prefixes of this snapshot. After tuning the parameters, a series of 20 snapshots of the week following that date was examined for the tuned architecture. Figures 9(a) and 9(b) show the prefix count variations of the whole routing table and length 24 during that week.

We expect a well adjusted architecture to waste few memory places in order to have a main table usage of nearly 100%. In the case of reconfigurable search tree, a nearly 100% usage implies the optimum resource usage. However it may overflow the tree and result in the rejection of further addition operations unless a full reconfiguration is done and the tree is enlarged.

6.1 Stability

Our reconfigurable search tree has a limited number of nodes. If it overflows, further prefix additions are not possible and a full reconfiguration is needed. We show if the architecture adjustment is performed properly, this case will be rare.

Starting from an initial set of parameter values, we set *MPR*=1.1 to make the main table slightly larger than the number of prefixes. In order to use the same ratio for the size of the reconfigurable search tree *RCR* is set to 1.1. The main table aspect ratio is set by *WHR*=0.1.

Note that these values are initialized once in order to be able to adjust the architecture according the first routing table snapshot. Then, other snapshots in sequence are examined on the architecture to see its behavior. Figure 10(a) shows variations of the main table usage and the reconfigurable search tree usage for these initial parameter values. To easily refer to this set of parameters and other following parameter sets, we summarized them in Table 1. As seen in the figure, after 4 days, the tree overflowed in the 12th routing table snapshot. It was expected due to the small value of *RCR*=1.1.

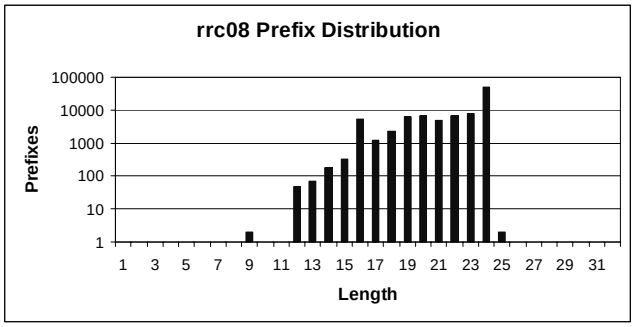
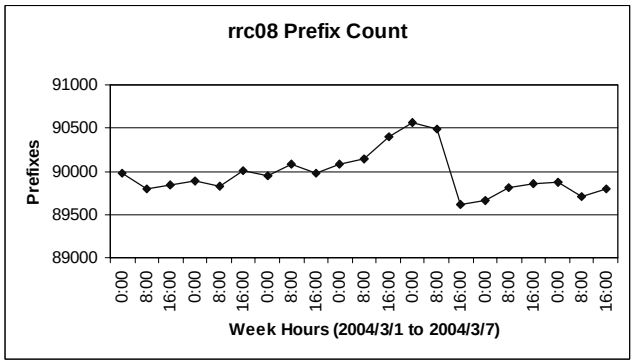
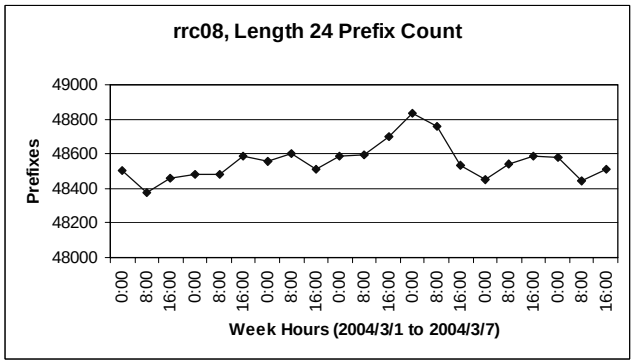


Figure 8. Prefix distribution of a snapshot of *rrc08* routing table.



(a)



(b)

Figure 9. Prefix count variations of (a) the whole routing table and (b) length 24 of *rrc08* during a week.

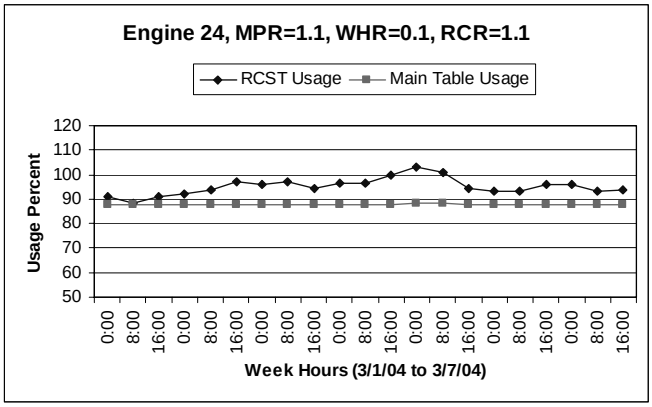
We assume a bigger *RCR*=1.5 value in the next parameter set (B). In addition, we increased *MPR* to 1.3 to allow more prefixes to settle on the main table. The result is shown in Figure 10(b). As seen in the graph, no tree overflows occurred in the one week duration. However, the tree usage is very sensitive to the prefix count variations shown in Figure 9(b). The 13th snapshot causes a peak in both graphs. Table 2 shows the cause of this sensitivity. This table shows the resource usage of the architecture for the different parameter sets. For parameter set B, the number of the nodes in the tree is 81 which is very small.

Table 1. Parameter sets used to evaluate the architecture.

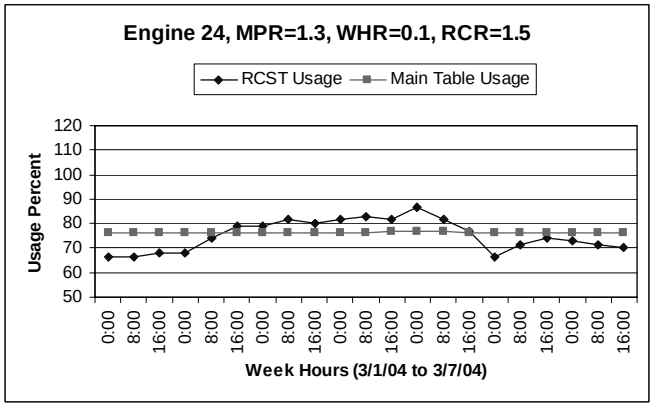
Parameter Set	MPR	WHR	RCR
A	1.1	0.1	1.1
B	1.3	0.1	1.5
C	1.0	0.1	1.5
D	1.3	0.05	1.5

To investigate the affect of a small value for *MPR*, the third parameter set with *MPR*=1.0 was chosen. As seen in Figure 10(c), this results in a near-full main table. The reconfigurable search tree has a safe distance from overflowing varying around 70% usage. However, Table 2 shows a big reconfigurable search tree as a result of this parameter set C which has implementation problems due to its large number of nodes.

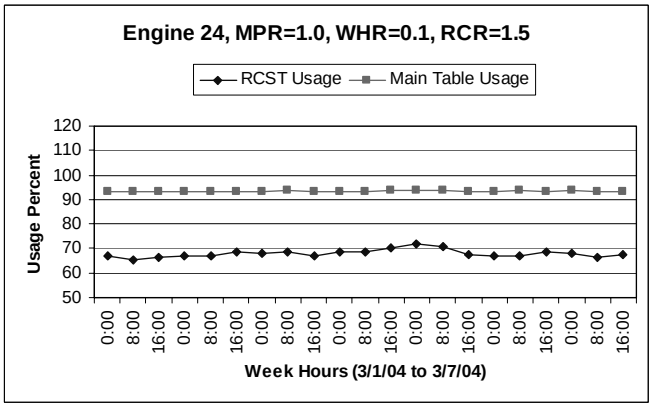
As the last experiment, the parameter set B was changed by reducing the previous value of *WHR* to half. This reduces the number of the main table columns and increases the number of rows. This is useful when the hardware platform offers bigger on-chip distributed memories and a lot of memory may be wasted when separate small memory blocks are needed. However, this increased the amount of collided entries by near 1% which is negligible in the graph. On the other hand, the number of reconfigurable search tree nodes was increased from 81 to 499 in Table 2. The increase in the number of tree nodes reduces the sensitivity of variations in the tree usage to the variations in the number of prefixes. Figure 10(d) shows that the tree usage varies safely below 70% in this case. This shows that it is possible to adjust the architecture to be stable against routing table updates so that no reconfigurable search tree overflows happens after routing table updates for a long period of time.



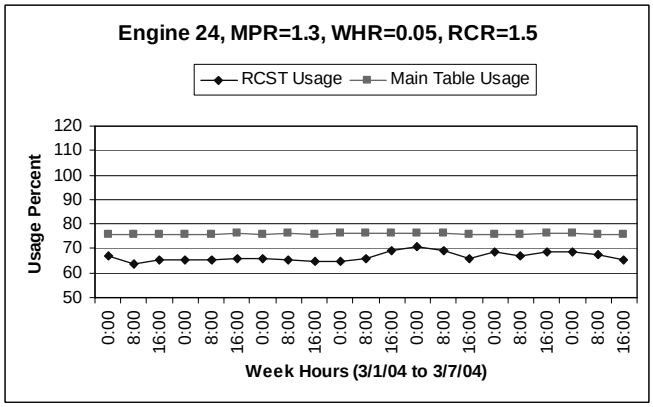
(a)



(b)



(c)



(d)

Figure 10. Variations of the reconfigurable search tree (RCST) usage and the main table usage of the engine of length 24 for the parameter values of Table 1.

6.2 Resource Usage

Table 2 shows the memory requirements of the architecture for prefixes of length 24 and the whole routing table for different parameter sets of Table 1. The number of nodes in the reconfigurable search tree which implies the amount of logic resources needed on the reconfigurable hardware platform is also shown in this table.

We synthesized the architecture according to the parameter sets B and D whose simulation showed better results. Xilinx Virtex-II Pro Family FPGA xc2vp100 [23] was used as the target reconfigurable hardware platform. This family of FPGAs offer distributed on-chip SRAM blocks of size 18 KBits. Therefore, all the main table columns were instantiated as a quantum of this size. This causes more memory usage which can be seen when comparing Table 3 with Table 2. FPGA slice utilization can also be seen in Table 3. This includes slices used for both search and update logic as well as the reconfigurable search trees.

Table 2. Resource requirements for the architectures with different parameters of Table 1.

Parameter Set	Memory (KBits)		Tree Nodes	
	Length 24	All Lengths	Length 24	All Lengths
A	1520	2650	970	3431
B	1778	3114	81	1161
C	1376	2403	3865	8775
D	1778	3124	499	1676

Table 3. FPGA resource utilization for synthesis with two different parameter sets.

Parameter Set	Block RAMs	Slices
B	276 (57%, 4968 KBits)	13950 (30%)
D	254 (52%, 4572 KBits)	14274 (31%)

6.3 Performance

Table 4 shows the performance of search and update operations for the synthesized pipelined architecture in terms of millions of operations per second (MOPS). Search pipeline depth is also reported. It is equal to the deepest engine depth plus the depth of the selection tree which is used to find the longest match among all engines match results.

Table 4. Search and update performance for the synthesized architecture.

Parameter Set	Search Throughput (MOPS)	Search Pipeline Depth	Main Table Update (MOPS)	Overall Update (MOPS)
B	269.03	12	88.47	2.786
D	263.23	13	85.97	1.946

If an update operation fails in the main table, the reconfigurable search tree must be updated using a partial reconfiguration to add or remove a prefix. To calculate the total time for prefix update, including this hardware reconfiguration overhead, we estimated the tree update probability proportional to the number of the prefixes in the tree. The target FPGA has a minimum partial reconfiguration time of 27 microseconds for a frame i.e. a column of logic resources inside the FPGA [23]. Although we only need to reconfigure a small part of an FPGA frame, we have to reconfigure at least one frame on this platform as the smallest reconfiguration quantum. A reconfigurable hardware platform with smaller reconfiguration quantum will result in a better update performance.

Table 5 compares our proposed approach with some other works. Memory requirements are expressed in terms of bits per prefix to have a fair comparison. In fact, we took into account only the prefixes in the main table to calculate the memory usage of our proposed architecture. It requires less memory than other schemes in Table 5. It requires less memory than parallel hashing scheme because the pointers to the subtables of each row of the main table are no longer needed.

Table 6 compares the lookup speed with some implementations. Unlike Table 5 which compares simulation results, synthesis results are compared in Table 6. Lookup performance is improved in our scheme even though we support a much larger routing database.

Table 5. The proposed architecture simulation results compared with others.

Method	Memory Required (KBits)	Prefix Count	Memory Required (Bits/Prefix)	Memory Accesses per Lookup
Proposed Scheme	3124	88303	35.38	1
DIR-24-8 [4]	264000	-	-	1-2
Trie Bitmap [13]	20000	41811	478.34	1-5
Indirect Lookup [5]	3600	40000	90	1-3
Parallel Hashing [9]	1512	37000	40.86	1-5
Multi-way Search [8]	5600	30000	186.67	1-9

Table 6. The synthesized architecture performance compared with others.

Implementation	Prefix Count	Lookup Speed (Mlps)
Proposed Architecture	89979	263.23
Reconfigurable BDD [3]	33796	157.7
Reconfigurable FSM [15]	38367	22.22
FIPL (Parallel Trie Bitmap) [18]	16564	9.09

7. CONCLUSION

A new reconfigurable architecture was proposed for IP address lookup. To reduce the resource usage, which is the problem of previously proposed reconfigurable hardware-based approaches, not all prefixes - but only the hash operation collisions - are implemented on the reconfigurable part of the system. This approach also causes only a small fraction of route updates to need hardware reconfiguration. This improves the update time. Having an IP lookup performance of 263 Mlps, this architecture can be used as a lookup engine for an Internet router forwarding packets at a speed over 250 Gbps.

8. ACKNOWLEDGMENTS

This project was supported in part by Iran Telecommunications Research Center (ITRC).

9. REFERENCES

[1] H. Chao, C. Lam, and E. Oki, "Broadband Packet Switching Technologies", John Wiley Publications, 2001, 365-405.

[2] K. Compton, and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", ACM Computing Surveys (CSUR), Vol. 34, No. 2, June 2002, 171-210.

[3] M. Desai, R. Gupta, A. Karandikar, K. Saxena, and V. Samant, "Reconfigurable Finite-State Machine Based IP Lookup Engine for High-Speed Router", IEEE Journal on Selected Areas in Communications (JSAC), Vol. 21, No. 4, May 2003, 501-512.

[4] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds", IEEE Informatics and Communications Conference (INFOCOM), Vol. 3, April 1998, 1240-1247.

[5] N. Huang, S. Zhao, J. Pan, and C. Su, "A Fast IP Routing Lookup Scheme for Gigabit Switching Routers", IEEE Informatics and Communications Conference (INFOCOM), Vol. 3, March 1999, 1429-1436.

[6] R. Jain, "A Comparison of Hashing Schemes for Address Lookup in Computer Networks", IEEE Transactions on Communications, Vol. 40, No. 10, October 1992, 1570-1573.

[7] F. Kuhns, J. DeHart, A. Kantawala, R. Keller, J. Lockwood, P. Pappu, D. Richard, D. Taylor, J. Parwatikar, E. Spitzangel, J. Turner, and K. Wong, "Design and Evaluation of a High-Performance Dynamically Extensible Router", IEEE DARPA Active Networks Conference and Exposition (DANCE), May 2002, 42-46.

[8] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search", IEEE/ACM Transactions on Networking (TON), Vol. 7, No. 3, June 1999, 324-334.

[9] H. Lim, J. Seo, and Y. Jung, "High Speed IP Address Lookup Architecture Using Hashing", IEEE Communications Letters, Vol. 7, No. 10, October 2003, 502-504.

[10] A. McAuley, P. Tsuchiya, and D. Wilson, "Fast Multilevel Hierarchical Routing Table Lookup Using Content Addressable Memory", US Patent, No. 05386413, January 1995.

[11] D. Morrison, "PATRICIA- Practical Algorithm to Retrieve Information Coded in Alphanumeric", Journal of ACM (JACM), Vol. 15, No. 4, October 1968, 514-534.

[12] S. Nilsson, and G. Karlsson, "IP-Address Lookup Using LC-Tries", IEEE Journal on Selected Areas in Communications (JSAC), Vol. 17, No. 6, June 1999, 1083-1092.

[13] D. Pao, C. Liu, A. Wu, L. Yeung, and K. Chan, "Efficient Hardware Architecture for Fast IP Address Lookup", IEEE Informatics and Communications Conference (INFOCOM), Vol. 2, June 2002, 555-561.

[14] M. Ruiz-Sánchez, E. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms", IEEE Network Magazine, March/April 2001, 8-23.

[15] R. Sangireddy, and A. Somani, "High-Speed IP Routing with Binary Decision Diagrams Based Hardware Address Lookup Engine", IEEE Journal on Selected Areas in Communications (JSAC), Vol. 21, No. 4, May 2003, 513-521.

[16] M. Sprachmann, "Automatic Generation of Parallel CRC Circuits", IEEE Conference on Design and Test of Computers, Vol. 18, No. 3, May 2001, 108-114.

[17] V. Srinivasan, and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion", ACM Transactions on Computer Systems (TOCS), Vol. 17, No. 1, February 1999, 1-40.

[18] D. Taylor, J. Lockwood, T. Sproull, J. Turner, and D. Parlour, "Scalable IP Lookup for Programmable Routers", IEEE Informatics and Communications Conference (INFOCOM), Vol. 2, 2002, 562-571.

[19] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups", ACM Special Interest Group on Data Communications (SIGCOMM), September 1997, 25-36.

[20] L. Wuu, and S. Pin, "A Fast IP Lookup Scheme for Longest-Matching Prefix", IEEE International Conference on Computer Networks and Mobile Computing (ICCNMC), October 2001, 407-412.

[21] N. Yazdani, and P. Min, "Fast and Scalable Schemes for the IP Address Lookup Problem", IEEE Conference on High Performance Switching and Routing (HPSR), 2000, 83-92.

[22] Routing Information Service, Online, http://www.ripe.net/ris/.

[23] Xilinx Virtex-II Pro FPGA Family Users Guide and Datasheets, Online, http://www.xilinx.com.