# IP-Lookup with a Blooming Tree Array: A New Lookup Algorithm for High Performance Routers

Gianni Antichi
gianni.antichi@iet.unipi.it

Andrea Di Pietro
andrea.dipietro@iet.unipi.it

Domenico Ficara
domenico.ficara@iet.unipi.it

Stefano Giordano
s.giordano@iet.unipi.it

Gregorio Procissi
g.procissi@iet.unipi.it

Cristian Vairo
cristian.vairo@iet.unipi.it

Fabio Vitucci
fabio.vitucci@iet.unipi.it

## ABSTRACT

Because of the rapid growth of both traffic and links capacity, the time budget to perform IP address lookup on a packet continues to decrease and lookup tables of routers unceasingly grow. Therefore, new lookup algorithms and new hardware platform are required. This paper presents a new scheme on top of the NetFPGA board which takes advantage of parallel queries made on perfect hash functions. Such functions are built by using a very compact and fast data structure called Blooming Trees, thus allowing the vast majority of memory accesses to involve small and fast on-chip memories only.

## Keywords

High Performance, IP Address Lookup, Perfect Hash, Bloom Filters, FPGA

## 1. INTRODUCTION

The primary task of a router is the IP-address lookup: it requires that a router looks, among possibly several thousands of entries, for the best (i.e., the longest) rule that matches the IP destination address of the packet. The explosive growth of Internet traffic and link bandwidth forces network routers to meet harder and harder requirements. Therefore, the search for the Longest Prefix Match (LPM) in the forwarding tables has now become a critical task and it can result often into the bottleneck for high performance routers. For this reason a large variety of algorithms have been presented, trying to improve the efficiency and speed of the lookup.

The algorithm here proposed is based on data structures called Blooming Trees (hereafter BTs) [8], compact and fast techniques for membership queries. A BT is a Bloom Filter based structure, which takes advantage of low false positive probability in order to reduce the mean number of memory accesses. Indeed, the number of required memory accesses is one of the most important evaluation criterion for the quality of an algorithm for high performance routers, given that it strongly influences the mean time required for a lookup process.

An array of parallel BTs accomplishes the LPM function for the entries of the forwarding table by storing the entries belonging to the 16–32 bit range. Every BT has been configured according to the Minimal Perfect Hash Function (MPHF) [1], a scheme conceived to obtain memory efficient storage and fast item retrieval. Shorter entries, instead, are stored in a very simple Direct Addressing (DA) logical block. DA module uses the address itself (in this case only the 15 most significant bits) as on offset to memory locations.

The implementation platform for this algorithm is the NetFPGA [12] board, a new networking hardware which proves to be a perfect tool for research and experimentation. It is composed of a full programmable Field Programmable Gate Array (FPGA) core, four Gigabit Ethernet ports and four banks of Static and Dynamic Random Access Memories (S/DRAM).

This work is focused on the data-path implementation of the BT-based algorithm for fast IP lookup. The software control plane has been also modified in order to accommodate the management and construction of the novel data structure. The software modifications merges perfectly in the preexistent *SCONE* (Software Component of the NetFPGA).

The rest of the paper is organized as follows: after the related work in address lookup area, section 3 illustrates the main idea, the overall algorithm and the data structures of our scheme. Then section 4 shows the actual implementation of our algorithm on NetFPGA while section 5 presents the modifications in the control plane software. Finally, section 6 shows the experimental results and section 7 ends the paper.

## 2. RELATED WORK

Due to its essential role in Internet routers, IP lookup is a well investigated topic, which encompasses trie-based schemes as well as T-CAM solutions and hashing

techniques. Many algorithms have been proposed in this area ([4][6][9][10][14][15]); to the best of our knowledge, the most efficient trie-based solutions in terms of memory consumption and lookup speed are Lulea and Tree Bitmap.

Lulea [4] is based on a data structure that can represent large forwarding tables in a very compact form, which is small enough to fit entirely in the L1/L2 cache of a PC Host or in a small memory of a network processor. It requires the prefix trie to be complete, which means that a node with a single child must be expanded to have two children; the children added in this way are always leaf nodes, and they inherit the next-hop information of the closest ancestor with a specified next-hop, or the undefined next hop if no such ancestor exists. In the Lulea algorithm, the expanded unibit prefix trie denoting the IP forwarding table is split into three levels in a 16-8-8 pattern. The Lulea algorithm needs only 4-5 bytes per entry for large forwarding tables and allows for performing several millions full IP routing lookups per second with standard general purpose processors.

Tree Bitmap [6] is amenable to both software and hardware implementations. In this algorithm, all children nodes of a given node are stored contiguously, thus allowing for using just one pointer for all of them; there are two bitmaps per node, one for all the internally stored prefixes and one for the external pointers; the nodes are kept as small as possible to reduce the required memory access size for a given stride (thus, each node has fixed size and only contains an external pointer bitmap, an internal next hop info bitmap, and a single pointer to the block of children nodes); the next hops associated with the internal prefixes kept within each node are stored in a separate array corresponding to such a node. The advantages of Tree Bitmap over Lulea are the single memory reference per node (Lulea requires two accesses) and the guaranteed fast update time (an update of the Lulea table may require the entire table to be almost rewritten).

A hardware solution for the lookup problem is given by CAMs, which minimize the number of memory accesses required to locate an entry. Given an input key, a CAM device compares it against all memory words in parallel; hence, a lookup actually requires one clock cycle only. The widespread use of address aggregation techniques like CIDR requires storing and searching entries with arbitrary prefix lengths. For this reason, T-CAMs have been developed. They could store an additional *Don't Care* state thereby enabling them to retain single clock cycle lookups for arbitrary prefix lengths. This high degree of parallelism comes at the cost of storage density, access time, and power consumption. Moreover TCAMs are expensive and offer little adaptability to new addressing and routing protocols [2].

Therefore, other solutions which use tree traversal and SRAM-based approach are necessary. For example, the authors of [11] propose a scalable, high-throughput SRAM-based dual linear pipeline architecture for IP Lookup on FPGAs, named DuPI. Using a single Virtex-4, DuPI can support a routing table of up to 228K prefixes. This architecture can also be easily partitioned, so as to use external SRAM to handle even larger routing tables, maintains packet input order, and supports in-place nonblocking route updates.

Other solutions take advantage of hashing techniques for IP lookup. For instance, Dharmapurikar et al. [5] use Bloom Filters (BFs) [3] for longest prefix matching. Each BF represents the set of prefixes of a certain length, and the algorithm performs parallel queries on such filters. The filters return a yes/no match result (with false positives), therefore the final lookup job is completed by a priority encoder and a subsequent search in off-chip hash tables. Instead, in our scheme, we will use BF-like structures which have been properly modified in order to directly provide an index for fast search.

## 3. THE ALGORITHM

All the algorithms previously described remark the most important metrics to be evaluated in a lookup process: lookup speed, mean number of memory access and update time. Each of the cited solutions tries to maximize general performance, with the aim of be implemented on a high performance router and obtain *line–rate* speed. The main motivations for this work come from the general limitations for high–performance routing hardware: limited memory and speed. Specifically, because of the limited amount of memory available, we adopt a probabilistic approach, thus reducing the number of external memory accesses also.

Because of the large heterogeneity of real IP prefixes distribution (as shown in several works as [5] and [13]), our first idea is to divide the entire rule database into two large groups, in order to optimize the structure:

- the prefixes of length $\leq 15$, which are the minority of IP prefixes , are simply stored in a Direct Addressing array; this solution is easily implemented in hardware and requires an extremely low portion of the FPGA logic area;
- the prefixes of length $\geq 16$ are represented by an array of Blooming Trees (hereafter called BT-array).

In the lookup process, the destination address under processing is hashed and the output is analyzed by the BT-array and the DA module *in parallel* (see fig. 1). Finally, an *Output Controller* compares the results of both modules and provides the right output (i.e., the longest matching), which is composed of a next-hop ad-

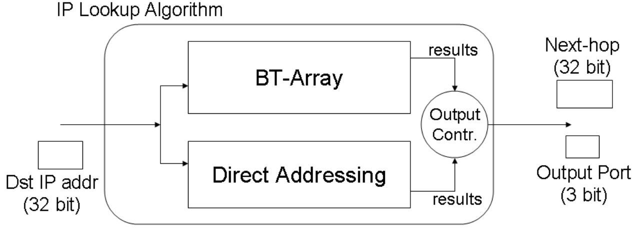



**Figure 1: The overall IP lookup scheme.**

dress (32 bits) and an output port number (3 bits, given that the NetFPGA has 8 output ports).

In the BT-array the prefixes are divided into groups based on their lengths and every group is organized in an MPHF structure (as shown in fig. 2). Therefore, the BT-array is an array where 17 parallel queries are conducted at the same time; at the end of the process, a bus of 17 wires carries the results: a wire is set to 1 if there is a match in the corresponding filter. Then a priority encoder collects the results of the BT-array and takes the longest matching prefix, while a SRAM query module checks the correctness of the lookup (since BTs are probabilistic filters in which false positives can happen).

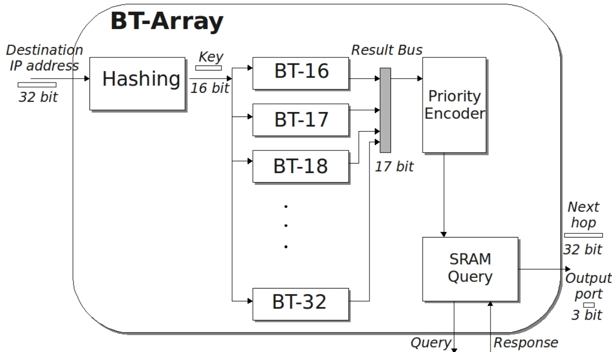



**Figure 2: BT-array schematic.**

## 3.1 Blooming Tree for MPHF

The structure we adopt to realize a Minimal Perfect Hashing Functions is a Blooming Tree [8], which is based on the same principles of Bloom Filters and allows for a further memory reduction. The idea of BT is constructing a binary tree upon each element of a plain Bloom Filter, thus creating a multilayered structure where each layer represents a different depth-level of tree nodes.

A Blooming Tree is composed of $L + 1$ layers:

- a plain BF ($B_0$) with $k_0$ hash functions $h_j$ ($j = 1 \dots k_0$) and $m$ bins such that $m = nk_0/\ln 2$ (in order to minimize the false positive probability);
- $L$ layers ($B_1 \dots B_L$), each composed of $m_i$ ($i = 1 \dots L$) blocks of $2^b$ bits.

Just as a BF, $k_0$ hash functions are used. Each of them provides an output of $\log_2 m + L \times b$ bits: the first group of $\log_2 m$ bits addresses the BF at layer 0, while the other $L \times b$ bits are used for the upper layers. The lookup for an element $\sigma$ consists of a check on $k_0$ elements in the BF (layer 0) and an exploration of the corresponding $k_0$ "branches" of the Blooming Tree.

"Zero-blocks" (i.e., blocks composed of a string of $b$ zeros which are impossible to be found in a naive BT for construction) are used to stop the "branch" from growing as soon as the absence of a collision is detected in a layer, thus saving memory. This requires additional bitmaps to be used in the construction process only. For more details about BTs, refer to [8].

Our MPHF on BT is based on the statement that, taken the BT as ordering algorithm (with $k_0$=1), a MPHF of an element $x \in S$ ($S$ is a set of elements) is simply the position of $x$ in the BT:

$$\mathrm{MPHF}(x) = \underset{S,BT}{\mathrm{position}}(x) \qquad (1)$$

All we need to care when designing this structure is that, in the construction phase, all the collisions vanish, in order to achieve a perfect function.

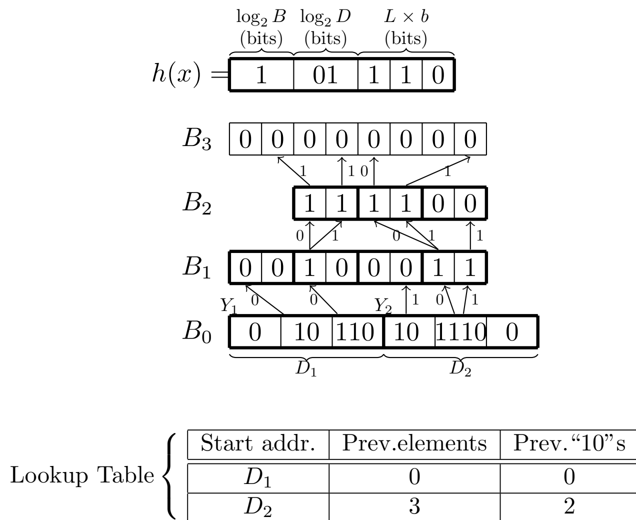
Instead, as for the lookup, the procedure that finds the position of an element $x$ is divided into two steps:

- find the tree which $x$ belongs to (we call it $T_x$) and compute the number of elements at the $T_x$'s left;
- compute the leaves at the left of $x$ in $T_x$.

In order to simplify the process, we propose the HSBF [7] as the first level of the BT, instead of the standard BF. The HSBF is composed of a series of bins encoded by Huffman coding, so that a value $j$ translates into $j$ ones and a trailing zero. Therefore, the first step of the procedure is accomplished by a simple popcount in the HSBF of all the bins at the left of $x$'s bin. As for the second step, we have to explore (from left to right) the tree $T_x$ until we find $x$, thus obtaining its position within the tree. The sum of these two components gives the hash value to be assigned. For more details about a MPHF realized by means of BT, refer to [1].

A simple example (see fig. 3) clarifies the procedure: we want to compute the MPHF value of the element $x$. In order to simplify the search in the HSBF, this filter is divided into $B$ sections of $D$ bins, which are addressed through a lookup table. Let us assume $B = 2$, $D = 3$, and $b = 1$: hence, the hash output is 6-bits long.

Let us suppose $h(x) = 101110$. The first bit is used to address the lookup table: it points to the second entry. We read the starting address of section $D_2$ and




| Start addr. | Prev.elements | Prev.“10”s |
|---|---|---|
| $D_1$ | 0 | 0 |
| $D_2$ | 3 | 2 |

**Figure 3: Example of hash retrieval through BT.**

that 3 elements are in the previous sections (i.e., already assigned by the MPHF). Now we use the next two bits of $h(x)$ to address the proper bin in section $D_2$: “01” means the second bin. The popcount on the previous bins in section $D_2$ indicates that another element is present (so far the total number of elements at $T_x$'s left is 4).

Then we focus on $T_x$: to move up to the next layer, we both use the third information in the table (the number of '10's in previous sections, which is 2) and count the number of “10”s in the previous bins of this section (that is 1). The sum shows that, before our bin, 3 bins are not equal to 0, so we move to the fourth block in layer $B_1$.

Here, the fourth bit of $h(x)$ allows to select the bit to be processed: the second one. But we want to know all the $T_x$'s leaves at $x$'s left, hence we have to explore all the branches belonging to the bin under processing. So we start from the first bit of the block and count the number of zero-blocks we find: 2, at layer $B_3$. Now the counter reads 6.

Regarding the second bit of the block (which is “the bit of $x$”), a popcount in layer $B_1$ indicates the third block in layer $B_2$: it is a zero-block, so we have found the block representing our element only: $x$ is the 7-th element in our ordering scheme. Then $\text{MPHF}(x) = 6$.

# 4. IMPLEMENTATION

## 4.1 MPHF Module

As above mentioned, the main component of the algorithm is the BT-array, which is composed of a series of MPHFs realized through BTs. Because of the large difficulties in allocating a variable–sized structure in hardware and for the sake of simplicity, in our implementation we simplify the scheme proposed in [1] and adopt a fixed-size structure. In details, the implemented structure presents 3 layers:

- Layer 0: a *Counting Bloom Filter* (CBF) composed of 128 *sections* and with 16 *bins* for every section;
- Layer 1: a simple bitmap that contains *two* bits for every bin of the level 0;
- Layer 2: another bitmap with *two* bits for every bit of the level 1; its size is then of 8192 bits.

These parameters (in terms of number of bins, sections and layers) are chosen in order to allocate, with a very low false positives probability, up to 8192 prefixes per prefix length, which implies that the total maximum number of entries is 128 thousands. Therefore, this implementation can handle even recent prefix rules databases and largely overcome the limitations of the simple (linear-search-based) scheme provided with the standard NetFPGA reference architecture.

Every bin of the CBF, according to the original idea in [1] and as shown in [7], is Huffman–encoded. Again, in order to simplify the hardware implementation, each bin consists of 5 bits and its length is fixed. Thus a maximum of 4 elements are allowed at level 0 for the same bin (i.e.: a trailing zero and max 4 bits set to 1). Since the probability of having bins with more than 4 elements is quite small (around $10^{-2}$) even if the structure is crowded, this implementation allows for a large number of entries to be stored.

Moreover, a lookup table is used to perform the lookup in the layer 0, which is composed of 128 rows containing the SRAM initial address for each section of the CBF. We place the entire BT and the lookup table in the fast BRAM memory: the CBF at layer 0 occupies a block of $2048 \times 5$ bits, while the lookup table has a BRAM block of $128 \times 20$ bits.

## 4.2 H3 Hash Function

The characteristics of the hash function to be used in the MPHF are not critical to the performance of the algorithm, therefore a function which ensures a fast hash logic has been implemented: the *H3* class of hardware hash functions.

Define $A$ as the *key* space (i.e. inputs) and $B$ as the *address* space (i.e. outputs):

- $A = 0, 1, ..., 2^i - 1$
- $B = 0, 1, ..., 2^j - 1$

where $i$ is the number of bits in the key and $j$ is the number of bits in the address. The H3 class is defined as follows: denote $Q$ as the set of all the $i \times j$ boolean matrices. For a given $q \in Q$ and $x \in A$, let $q(k)$ be the

$k$–th row of the matrix $q$ and $x_k$ the $k$–th bit of $x$. The hashing function $h_q(x): A \rightarrow B$ is defined as:

$$h_q(x) = x_1 \cdot q(1) \oplus x_2 \cdot q(2) \oplus \ldots \oplus x_i \cdot q(i) \quad (2)$$

where $\cdot$ denotes the binary AND operation and $\oplus$ denotes the binary exclusive OR operation. The class H3 is the set $\{h_q | q \in Q\}$.

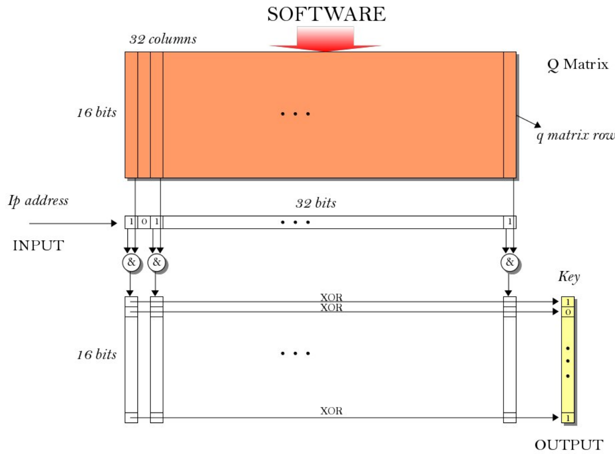



**Figure 4: The scheme of the hash function belonging to the H3 class. The Q matrix is stored in a BRAM block.**

Figure 4 shows our H3 design. The process is divided into two steps: first the 32-bit input value (i.e. the IP address) is processed by a block of AND, then the resulting 32 vectors of $n$ bits are XOR-ed by a matrix of logical operations and a resulting string of $n$ bits is provided as output.

The length of the output is chosen to meet the requirements of the following BT-array. The $q$ matrix is pre–programmed via software, passed through the PCI bus, and stored in a BRAM block.

## 4.3 Managing false positives

As already stated, a BT provides also a certain amount of false positives with probability $f$. Thus every lookup match has to be confirmed with a final lookup into SRAM. Then, intuitively, the average number of SRAM accesses $\overline{n}$ increases as $f$ grows. More formally, assuming all BTs in the BT-array have the same false positive probability $f$, we can write:

$$\overline{n} \leq 1 + \sum_{i=1}^{16} f^i \leq \frac{1}{1-f} \quad (3)$$

This equation takes into account the probability of the worst case that happens when all BTs provide false positives and are checked in sequence. As one can easily verify, even if $f$ is quite large, the average number of memory accesses is always close to 1 (less than 1.11 for $f = 0.1$).

# 5. CONTROL PLANE

The MPHF IP lookup algorithm needs a controller that manages the building of the database, the setup of the forwarding table, and the potential updates of the prefixes structures. All these functions have been implemented in C/C++ and integrated into the Software SCONE.

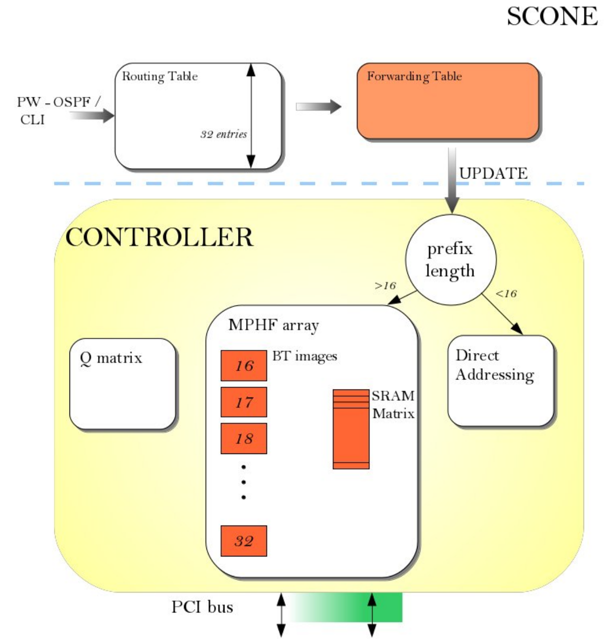



**Figure 5: In this figure the different functions of the software control plane can be seen.**

This software adopts PW-OSPF as routing protocol, which is a greatly simplified link–state routing protocol based on OSPFv2 (rfc 1247). Every time an update in the routing table occurs, a new forwarding table is created and passed entry by entry through the PCI bus to the NetFPGA. An entry in the forwarding table is composed of destination IP address, subnet mask, gateway, and output port. The original behavior in SCONE was to retransmit all the entries of the forwarding table for each update, while we transmit the modified (new, updated or deleted) entries only. Then the developed Controller analyzes these entries and modifies the proper structures. This communication takes advantage of the register bus.

## 5.1 Updates

When a modification in the forwarding table occurs, it may happen that the new elements lead to collisions in one of the MPHF structures (because they are limited to 2 layers only). In this case, the hash function has to be modified in order to avoid these collisions. This requires the change of the $Q$ matrix and its re-

transmission to the H3 hashing module, which stores the matrix in a BRAM block. The same communication protocol used for the construction is adopted, indeed a simple registers call has to be made. Therefore, a new hash function is used and all the data-structures for lookup are updated, thus obtaining MHPFs with no collisions.

# 6. RESULTS

In this section, the simulative results about the implementation of our algorithm are shown. In details, we focus on resource utilization, in terms of slices, 4-input LUTs, flip flops, and Block RAMs. We compare our results with those of the NetFPGA reference router where a simple linear search is implemented for IP lookup.

Table 1 shows the device utilization (both as absolute and relative figures) for the original NetFPGA lookup algorithm. It provides a simple lookup table which allows to manage 32 entries only to be looked for through a linear search. Instead we implement a more efficient and scalable algorithm, which is capable of handling up to 130000 entries (by assuming a uniform distribution for entries prefix length). This complexity is obviously paid in terms of resource consumption (see tab. 2): in particular, our lookup module uses 41% of the available slices on the Xilinx Virtex II pro 50 FPGA and 29% of the Block RAMs. However, as for the synthesis of the project, it is worth noticing that even though we use a wide number of resources, the timing closure is achieved without any need to re-iterate the project flow.

**Table 1: Resource utilization for the original lookup algorithm.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 935 out of 23616 | 3% |
| 4-input LUTS | 1321 out of 47232 | 2% |
| Flip Flops | 343 out of 47232 | 0% |
| Block RAMs | 3 out of 232 | 1% |

**Table 2: Utilization for our algorithm.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 9803 out of 23616 | 41% |
| 4-input LUTS | 10642 out of 47232 | 22% |
| Flip Flops | 19606 out of 47232 | 41% |
| Block RAMs | 68 out of 232 | 29% |

Tables 3 and 4 list the specific consumption of the main modules composing our project. In particular, the final synthesis summary is reported for both a single MPHF and the hashing module. It is interesting to observe the number of slices required for the hashing compared to that of the table 3: the hashing module ends up to be bigger than the whole MPHF module because of its complexity, since it has to provide 17 different hash values for each of the 17 MPHF blocks.

**Table 3: Utilization for a single MPHF.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 398 out of 23616 | 1% |
| 4-input LUTS | 561 out of 47232 | 1% |
| Flip Flops | 618 out of 47232 | 1% |
| Block RAMs | 6 out of 232 | 2% |

**Table 4: Utilization for the hashing module.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 1179 out of 23616 | 4% |
| 4-input LUTS | 1293 out of 47232 | 2% |
| Flip Flops | 1841 out of 47232 | 3% |

Finally, table 5 presents the overall device utilization for the reference router including our lookup algorithm and highlights the extensive use of the various resources. In particular we use 94% of the available Block Rams and 74% of slices and LUTs.

**Table 5: Utilization for our overall project.**

| Resources | XC2VP50 Utilization | Utilization Percentage |
|---|---|---|
| Slices | 17626 out of 23616 | 74% |
| 4-input LUTS | 32252 out of 47232 | 74% |
| Flip Flops | 31512 out of 47232 | 66% |
| Block RAMs | 220 out of 232 | 94% |
| External IOBs | 360 out of 692 | 52% |

# 7. CONCLUSIONS

This paper presents a novel scheme to perform longest prefix matching for IP lookup in backbone routers. By following the real IP prefixes distributions, we divide the entire rule database into two large groups, in order to optimize our scheme. For prefixes of length $< 16$, which are the minority of IP prefixes, we use a simple Direct Addressing scheme, while for the others we use an array of Blooming Trees.

The implementation platform for this work is the NetFPGA board, a new networking tool which proves to be very suitable for research and experimentation.

NetFPGA, originally, provides a simple lookup scheme which allows to manage 32 entries only by means of linear searches. Instead, our scheme is capable of handling up to 130000 entries at the cost of a bigger resource consumption. Anyway, the timing closure is achieved without any need to re-iterate the project flow.

## 8. ADDITIONAL AUTHORS

## 9. REFERENCES


[1] G. Antichi, D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Blooming trees for minimal perfect hashing. In *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*, pages 1–5, 30 2008-Dec. 4 2008.

[2] F. Baboescu, S. Rajgopal, L. B. Huang, and N. Richardson. Hardware implementation of a tree-based ip lookup algorithm for oc-768 and beyond. In *DesignCon*, 2006.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *Proc. of the ACM SIGCOMM '97*, pages 3–14, New York, NY, USA, 1997. ACM.

[5] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using bloom filters. In *SIGCOMM 2003*.

[6] W. Eatherton, Z. Dittia, and G. Varghese. Tree bitmap: Hardware/software ip lookups with incremental updates. In *ACM SIGCOMM Computer Communications Review*, 2004.

[7] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Multilayer compressed counting bloom filters. In *Proc. of IEEE INFOCOM '08*.

[8] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Blooming trees: Space-efficient structures for data representation. In *Communications, 2008. ICC '08. IEEE International Conference on*, pages 5828–5832, May 2008.

[9] P. Gupta and N. Mckeown. Packet classification using hierarchical intelligent cuttings. In *in Hot Interconnects VII*, pages 34–41, 1999.

[10] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proc. of SIGCOMM*, pages 203–214, 1998.

[11] H. Le, W. Jiang, and V. K. Prasanna. Scalable high-throughput sram-based architecture for ip-lookup using fpga. In *International Conference on Field Programmable Logic and Applications*, 2008.

[12] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[13] *Route Views 6447, http://www.routeviews.org/.*

[14] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting, 2003.

[15] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proc. of SIGCOMM*, pages 135–146, 1999.