LayeredTrees: Most Specific Prefix-Based Pipelined Design for On-Chip IP Address Lookups

Yeim-Kuan Chang, Fang-Chen Kuo, Han-Jhen Kuo, and Cheng-Chien Su

Abstract—Multibit trie-based pipelines for IP lookups have been demonstrated to be able to achieve the throughput of over 100 Gbps. However, it is hard to store the entire multibit trie into the on-chip memory of reconfigurable hardware devices. Thus, their performance is limited by the speed of off-chip memory. In this paper, we propose a new pipeline design called *LayeredTrees* that overcomes the shortcomings of the multibit trie-based pipelines. LayeredTrees pipelines the multi-layered multiway balanced prefix trees based on the concept of most specific prefixes. LayeredTrees is optimized to fit the entire routing table into the on-chip memory of reconfigurable hardware devices. No prefix duplication is needed and each W-bit prefix is encoded in a $(W+1)$-bit format to save memory. Assume the minimal packet size is 40 bytes. Our experimental results on Virtex-6 XC6VSX315T FPGA chip show that the throughputs of 33.6 and 120.8 Gbps can be achieved by the proposed single search engine and multiple search engines running in parallel, respectively. Furthermore, the impact of update operations on the search performance is minimal. With the same FPGA device, an IPv6 routing table of 290,503 distinct entries can also be supported.

Index Terms—IP lookup, most specific prefix, FPGA

1 Introduction

An IP router, sometimes called layer-3 switch, performs the forwarding decision for each incoming packet. All the operations executed by the router after receiving packets can be divided into time-critical (fast path) and non-time-critical (slow path) operations depending on the packet type. These time-critical operations run for most of the packets include IP packet validation, packet lifetime control, checksum recalculation, destination address parsing, and IP table lookups, as described in [2]. Among all the time critical operations, the IP table lookups are the most time consuming. Thus, IP table lookups must be implemented in a highly efficient fashion to keep up with the high-speed and high-bandwidth links connected between routers. IP table lookups are executed by a *forwarding engine* that looks up the packet's destination IP address against a *forwarding table* in order to determine the address of the next-hop router and the egress port to which the packet should be sent. The forwarding table must maintain an entry for every network allocated with an address block represented as a *route prefix*. Currently, each prefix can be of any length from 8 to 32. As a result, the search in a forwarding table can no longer be performed by exact matching. The IP table lookup becomes the *Longest Prefix Match* (LPM) problem which determines the longest route prefix covering the destination IP address of the packet because there may be one or more prefixes that match the destination address. Router designers are challenged to come up with fast algorithms or architectures for solving LPM.

There are so many schemes proposed for IP lookups in the literature [8], [9], [12], [36]. They can be classified into two categories: static and dynamic lookup schemes. An IP lookup scheme is called static if reconstruction is needed when an update occurs; otherwise, it is called dynamic. Static lookup schemes are acceptable when the forwarding table is not updated frequently. A forwarding table pre-computation is typically needed in static schemes for improving lookup speed and reducing memory usage. However, when prefixes are frequently added in or deleted from the forwarding table, a dynamic lookup scheme is needed. Many dynamic lookup schemes [6], [9], [16], [19], [29]-[32], [37], [41] were proposed. However, they are designed with the software-based routers in mind [33] and thus most of the performance results are obtained by the simulation on personal computers. Although these dynamic data structures can achieve better update performance than static ones, it is unknown that we can have an efficient hardware design for them in order to support the throughput of beyond 100 Gbps for large routing tables. On the other hands, a number of proposals have focused on exploiting the pipelining architectures. In this respect, the multibit-trie based pipeline architectures have gained much attention because each trie level can be directly mapped onto a pipeline stage with its own memory and uncomplicated processing logic. As a result, one IP lookup can be performed every clock cycle and thus the throughput can be significantly improved. Noticeable trie-based pipeline designs include Ring [1], CAMP [23], OLP [20], BiOLP [24], BPFL [13], POLP [13], [22], Bit-Shuffled Trie (BSTrie) [34], FlashLook [3], and FlashTrie [4]. These pipeline designs can normally achieve a much higher throughput than dynamic schemes based on software. For example, the throughputs achieved by BiOLP, FlashLook, and FlashTrie are beyond 100 Gbps. However, these multibit-trie-based hardware implementations exhibit

• *The authors are with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 70146, Taiwan. E-mail: {ykchang, p7895107, p7697102, p7894104}@mail.ncku.edu.tw*

Manuscript received 13 June 2012; revised 12 Apr. 2013; accepted 24 Apr. 2013. Date of publication 05 May 2013; date of current version 12 Nov. 2014. Recommended for acceptance by M. Guo
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.109

two disadvantages. First, the required memory for large routing tables is too large to fit into the on-chip memory of the most advanced reconfigurable devices such as FPGA. Thus, off-chip SRAM or DRAM are used to store the entire routing tables. Thus, the ultimate performance of these implementations is bounded by the speed of off-chip memory which is usually 5 to 8 ns for a single memory access. Second, the update performance will be slow. Since one single prefix of shorter length may need to be expanded to multiple prefixes of longer lengths, it is not easy to maintain the status of the original routing table and thus prefix deletions may be difficult to perform. Other than the trie-based approaches, DMST [11] was a pipeline design that is not based on trie. DMST is based on the B-tree data structure constructed from the endpoints of the prefixes. Extended version of DMST can also achieve the throughput of 100 Gbps. However, it has the same drawback as the trie-based pipelines that the required memory is too large to be fit into the on-chip memory of FPGA devices for large routing tables.

In this paper, we shall propose a high-speed pipelined architecture called *LayeredTrees*. It uses layer-based prefix partitioning scheme to construct the multiway B-tree data structure. We will show that, other than trie and range based data structure, *LayeredTrees* can achieve a throughput of 120 Gbps that is even higher than the best multibit-trie-based implementation nowadays. Such high throughput obtained by the proposed pipelined architecture comes from the non-duplication property of the underlined dynamic data structure along with the most memory-efficient representation to store prefixes. The data structure for the large routing table is small enough to be entirely fit in the on-chip memory of the FPGA devices. As a result, the ultimate performance of the proposed pipeline will not be bounded by the speed of off-chip SRAM or DRAM. In addition, since the proposed pipeline architecture is based a data structure originally designed for fast insertion and deletion, the impact of the update operations on the search performance is minimal.

The rest of the paper is organized as follows: The related work is discussed in section II. The dynamic multi-layered multiway balanced prefix trees called LayeredTrees and the memory reduction optimizations are described in section III. In section IV, the pipelined architecture proposed for LayeredTrees is described in details. All the analysis and experimental results are shown in section V and finally, the concluding remarks are presented.

2 Related Work

In this section, we first review the existing designs that are able to provide the throughput of more than 40 Gbps (OC-768). Then, we summarize the design model used for the proposed pipelined architecture.

2.1 Trie Based Pipeline Architecture

A trie data structure is a natural way to store prefixes. It is a tree-like data structure that directs the search for LPM in a bitwise fashion by using the destination IP address of the incoming packet. Trie is mostly implemented using linked lists in which each trie node has left and right pointers pointing to its left and right subtries, respectively.

The idea of trie based pipeline architectures is to map each level of the trie onto a pipeline stage with its own memory and processing logic. As a result, a lookup request can be completed every clock cycle and thus the overall throughput is greatly improved. However, this trie-based pipeline design results in unbalanced memory distribution over the pipeline stages because the memory required in lower trie levels is obviously much larger than that in higher trie levels. The memory unbalance in turn leads to a slower clock rate and a notable negative impact on route update and memory allocation.

The memory unbalance problem of the trie-based pipeline designs was first mentioned in [17], [39] where authors think using large amounts of cheap off-chip DRAM inefficiently is advantageous as long as a faster, simpler, and cheaper solution is realized. Also mentioned in [39], DRAM memory access times will become a bottleneck when the link speeds increase. At the link speeds of 40Gbps and beyond available today, the entire IP routing tables will need to be stored in SRAM or on-chip memory. Authors in [1] investigated the relationship between the on-chip SRAM memory access time and the size of memory required per stage. Their estimated results obtained by the memory generator application CACTI [38] show that the memory access time increases significantly with the size of memory.

Rather than based on the levels of leaf-pushed unibit tries, an alternative height-based pipeline design called *scalable dynamic pipelining* (*SDP*) is proposed in [18]. This work guarantees a tighter worst-case per-stage memory bound than previous level-based approaches. It ensures the number of leaves in the leaf-pushed unibit trie is equal to the number of prefixes by using a technique called jump nodes to limit the number of copies of a leaf-pushed node. As a result, each route-update requires only one write dispatched into the pipeline.

As proposed by Basu and Narilkar [5], the memory imbalance can be solved by minimizing the stage that has the largest memory in order to improve the clock rate of the bottleneck stage. In [5], the disruption of the fast path lookup pipeline by route updates is also minimized by reducing the number of write bubbles that are sent to the pipeline for updates. The write bubbles can be generated by software preprocess for prefix insertions and deletions. A *ring* like pipeline architecture is proposed in [1] to further reduce the memory imbalance problem. Any sub-trie of the search data structure is not restricted to be mapped to stages starting at the first stage. In other words, the root of each sub-trie can be mapped to any stage and descendents of the root are mapped to the later stages or wrapped around to form a ring. The drawback is that the maximum search throughput becomes a half of the pipeline stage clock rate, due to the memory access conflicts caused by this design. Furthermore, the pipeline architecture called *Circular, Adaptive and Monotonic Pipeline* (*CAMP*) is proposed in [23] to extend the above ring pipeline. CAMP decouples the numbers of pipeline stages and trie levels to give more freedom to map the sib-tries to pipeline stages. CAMP uses several initial bits as the hashing index to partition the trie into sub-tries and allows the roots of sub-tries to be mapped to any stage. To seek a balanced mapping of nodes onto pipeline stages, dynamic entry and exit points are allowed to exploit which requests can enter and exit at any stage. Since CAMP allows the requests to enter and exit at any stage, input and output FIFOs are needed to maintain the order of incoming packets, which make CAMP more complicated.

Three drawbacks caused by the multiple-entry and multiple-exit stages of CAMP are as follows. First, the request queues result in extra delay which leads to a large variation of total delay for each request. Second, due to the multiple-entry stages, the write bubbles disrupt the pipeline operations for route-updates. Finally, multiple-entry stages also cause access conflicts and thus reduce the throughput. Therefore, authors in [20] proposed a new mapping scheme called *Optimized Linear Pipeline* (*OLP*) that reverts to the linear pipeline. In OLP, all subtries' roots are mapped to the first stage. However, to achieve a more balanced memory usage, OLP allows the nodes on the same level of a subtrie to be mapped onto different pipeline stages, augmented by the no-operation (nop) instruction going through the stages. OLP uses a heuristic to map nodes to stages. Each balanced pipeline holds $\lceil N/(P-1) \rceil$ nodes except possibly the first stage, where N is the total number of nodes and P is the number of stages. The nodes of each sub-trie are stored into a segmented queue, where the nodes on the i^{th} level of the subtrie are stored in the i^{th} segment of the segmented queue. When the mapping process runs in stage j from $j = 1$ to $P - 1$ (stage 0 for the root nodes of all sub-tries), the following steps are executed. All the segmented queues are sorted in decreasing order of the number of segments in each segmented queue and also in decreasing order of the number of nodes in their frontend segments for a tie breaker. Only the nodes in frontend segments are popped from queues in order to be put in stage j. If all the nodes in the frontend segment of a queue have been popped, then the nodes in the next segment of the queue are not popped until the next stage $j + 1$. In order to further increase the flexibility to map trie nodes onto pipeline stages, a dual-entry bidirectional pipeline called *bidirectional optimized linear pipeline* (*BiOLP*) was designed in [24]. BiOLP employs the dual-port functionality of the on-chip Block RAMs to allow two simultaneous accesses to the local memory in each stage by the two search operations flowing in opposite directions. BiOLP is different from OLP in that some sub-tries are inverted by various proposed heuristics. The normal sub-tries are called forward sub-tries and the inverted ones are called reverse sub-tries. The priority of a node is defined as its height and its depth in the forward and reverse subtries, respectively. In other words, the nodes of upper levels in forward sub-tries and that of deep levels in reverse sub-tries have high higher priority. Thus, nodes with higher priority are mapped to current stage before that of lower priority. When the priority of a node is equal to the number of the remaining stages, it is regarded as a critical node and so it must be mapped onto the current stage before other nodes. BiOLP also maintains the packet input order and supports non-blocking route update. Moreover, BiOLP employs packet caching to improve the throughput. BiOLP is further improved in [21] by adopting multiple pipelines to facilitate processing multiple packets per clock cycle along with the IP caching and a lightweight scheduler and several small output delay queues.

A set of designs [25], [26], [28] were proposed to build a binary search tree for each group of disjoint prefixes using a level-based prefix partitioning scheme on the binary trie, which is more general than the partitioning scheme based on prefix lengths. A bound for the number of binary search trees is set to limit the number of pipeline needed. With the binary search trees, the pipeline designs can be implemented easily. Dual linear pipelined architectures with dual-ported memory are used to achieve high throughput of beyond 100 Gbps verified by the implementations on ASIC and FPGA. Another scheme improving on OLP called Parallel Optimized Linear Pipeline (POLP) architecture was also proposed in [22]. POLP is enhanced by pipelined prefix caches. Another parallel search hardware design based multi-bit tries called *Parallel Frugal Lookup* (*PFL*) was proposed in [14] Since the level modules designed in PFL is very complex compared to the traditional multi-bit trie nodes, PFL's cycle time is expected to be longer than the multibit trie based pipeline designs described above. In PFL, the next-hop information is stored in the external memory, while the structure of the lookup table is stored in on-chip memory. In [13], an enhanced version of PFL called *balanced parallelized frugal lookup* (*BPFL*) algorithm was proposed. The experimental results on Altera's Stratix II EP2S180F1020C5 FPGA chip showed that there is no significant performance difference in achieved throughput between BPFL and BiOLP. However, BPFL's total on-chip memory requirements are significantly lower than POLP. In [34], a binary trie based scheme called *bit-shuffled trie* (*BSTrie*) using a bit-shuffling technique was proposed to restructure the binary-trie in order to reduce the memory usage per prefix. Although BSTrie achieves the best memory utilization, its search speed becomes slower than some of the existing schemes due to its complex logic for lookup engine.

Multiple hashing function based designs can also be used to improve the throughput such as *FlashLook* [3], *FlashTrie* [4], and the multi-hash scheme [15], just to name a few. However, there are two disadvantages to use hashing. One is the collisions that make the pipeline un-deterministic in terms of delays required for finishing a lookup. The other is the rehashing that is needed during route updates. Both *FlashLook* in [3] and *FlashTrie* in [4] achieve high throughputs on off-chip DRAMs. Also, FlashTrie is based on multibit tries along with hashing functions.

2.2 Segment Tree Based Hardware Architecture

Other than the numerous multibit trie based pipeline designs, binary or multiway search trees based [11], [40] on the endpoints of prefixes can be developed. Two endpoint based pipeline designs, i.e., *pipelined dynamic segment tree* (*pDST*) [43] and *dynamic multiway segment tree* (*DMST*) [11] are proposed in the literature. The basic tree structure of pDST is 2-3 tree which is implemented as a bi-directional linear pipeline using dual-ported block RAMs. On the other hand, DMST is based on the B-tree and targets on using off-chip memory for storing the large routing tables.

2.3 Design Model

The design model for the proposed IP lookup algorithm is illustrated by Fig. 1. The protocol stack contains two planes, data plane and control plane. The control plane usually managed by a standard RISC processor consists of a large number of sophisticated codes that implement the slow-path protocols. The fast-path functions executed in data plane typically are the protocols in layer two or three of the network protocol stack. In this paper, we assume that the slow-path

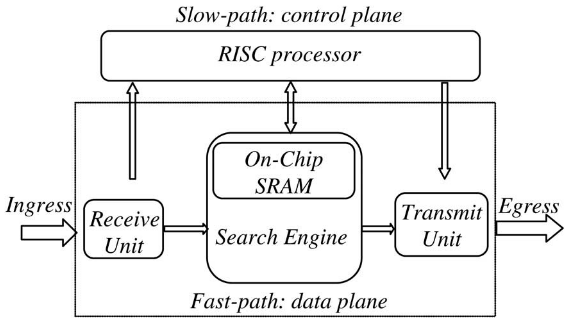

Fig. 1. Design model.

Table 1
Prefix Enclosure Analysis

Routing Table		AS6447 (2005-4)	AS6447 (2009-7)	AS6447 (2011-5)
size		163,535	301,552	369,394
Layer	0	150,245 (91.9%)	273,944 (90.8%)	334,445 (90.5%)
	1	11,135 (6.8%)	23,171 (7.7%)	28,930 (7.8%)
	2	1,775 (1.1%)	3,690 (1.2%)	4,870 (1.3%)
	3	329 (0.2%)	628 (0.2%)	926 (0.3%)
	4	45	101	177
	5	6	16	30
	6	0	2	12
	7	0	0	4

RISC processor mainly executes the route update operations and the IP lookups in fast-path are based on the proposed LayeredTrees data structure stored in the on-chip memory of the dedicated FPGA-based search engine. In Fig. 1, packets are received at the *receive unit* and transmitted to outside by *transmit unit* after the nexthop is determined. If the received packets are for updates, they are sent to the slow-path RISC processor. The processor executes the update operations and modifies the contents of the on-chip memory accordingly. If arrived packets are usual Internet packets, they are sent to search engine that executes the fast-path function to find the nexthop and finally are passed to the transmit unit.

3 Multi-Layered Multiway Prefix Trees

To sustain the information overload on the Internet nowadays, the very high-speed backbone routers with less memory consumption and more flexibility for updating routing tables are required. To speed up the search speed, a pipelined hardware design is considered in this paper. The dynamic multiway segment tree based scheme [11] implemented in hardware suffers from the 8 ns off-chip memory access time because its memory usage is larger than the on-chip memory available in today's most advanced FPGA devices. To avoid the memory access delay of off-chip memory, the entire data structure should be completely stored in the on-chip memory for improving the throughput of search engine. Although the scheme proposed in [40] is able to store the entire routing table in on-chip memory, time-intensive precomputation is needed for the proposed compressed data structure and thus dynamic updates are not supported.

In this paper, a data structure called *LayeredTrees* is proposed for dynamic routing tables. In LayeredTrees, the prefixes in a routing table are grouped into layers based the concept of the most specific prefixes (also called *layer grouping constraint*) proposed in our previous work [8]. In this paper, a multiway prefix tree structure along with four memory and pipeline logic optimizations will be proposed to reduce the memory requirement and increase the pipeline clock rate. The four optimizations include (1) the simple one-level leaf push scheme to reduce the number of layers required and thus the hardware cost, (2) the routing table split to reduce the number of bits needed for the prefixes of lengths 9 to 24 stored in B-tree nodes, (3) B-tree order varying scheme to choose the B-tree order as smaller as possible for the B-tree nodes in each layer, and (4) the stage 4 clock rate improving scheme to increase the clock rate of the bottleneck stage 4 of the proposed pipelines by breaking the stage 4 into three sub-stages. Our simulation results will show that the proposed design can achieve the throughput of 120 Gbps by using the FPGA device currently available.

All the most specific prefixes (i.e., the prefixes associated with the leaf nodes of the binary trie) that do not enclose any other prefixes are grouped into layer 0. Assume the prefixes in layer 0 are removed from the routing table. Now, the most specific prefixes are grouped into layer 1. The prefix grouping process can be performed recursively until no prefix exists. In general, any layer-i prefix encloses at least one layer-($i-1$) prefix but does not enclose prefixes in the higher layer for $i > 0$. The prefix enclosure analysis for various routing tables of router AS6447 [7] is shown in Table 1. There are at most eight layers. Besides, the most noticeable property is that all the prefixes in the same layer are disjoint. Thus, the prefixes in a layer can be sorted (described later) and then binary search can be applied to find the only matched prefix in the layer against the incoming destination IP address. The search process starts from the layer 0. If a matched prefix is found, this must be the longest matched prefix and thus the search process is finished. Otherwise, the search is performed on the prefixes in layer 1. This search process repeats until a match is found or the prefixes in the last layer are exhausted. Since layer 0 occupies more than 90% of the prefixes, most of the matches will be found after layer 0 is searched.

The goal of LayeredTrees is to store the entire routing table in the on-chip memory of FPGA device currently available. The prefixes in each layer are organized as a B-tree. Given a small example routing table as shown in Fig. 2, three layers can be constructed, layer-0 $\{P_7, P_{10}, P_{11}, P_5, P_8, P_{12}, P_9, P_3\}$, layer-1 $\{P_4, P_6, P_1\}$, and layer-2 $\{P_2\}$. Fig. 3 shows the three corresponding B-trees in the increasing order of the prefixes represented by the lite prefix format described below.

There are two problems when the B-tree is used to organize the prefixes in each layer. First, no matter how we construct the multi-layered B-trees by inserting prefixes in a sequential order or randomly, the utilization of the B-tree nodes is only about 50 percent, i.e., a half amount of storage for keys in the node are unused. Second, the pointers used for the branches of a B-tree node consume a large amount of memory. The first problem is solved by controlled B-tree building algorithm (CBA) proposed in this paper for improving the utilization of B-tree nodes and also for keeping sufficient slots to accommodate the newly inserted prefixes for efficient updates. For the second problem, we replace the branch pointers in an ordinary B-tree node by a base pointer pointing to the prebuilt aggregate array of B-tree nodes. Based on the number of

ID	Prefix	Nexthop
P_1	10*	NH_1
P_2	001*	NH_2
P_3	1011*	NH_3
P_4	0000*	NH_4
P_5	0001*	NH_5
P_6	0010*	NH_6
P_7	000000	NH_7
P_8	001011	NH_8
P_9	00111*	NH_9
P_{10}	000010	NH_{10}
P_{11}	000011	NH_{11}
P_{12}	001100	NH_{12}

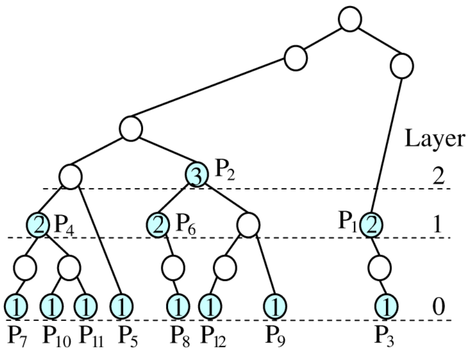

Fig. 2. An example routing table and its binary trie.

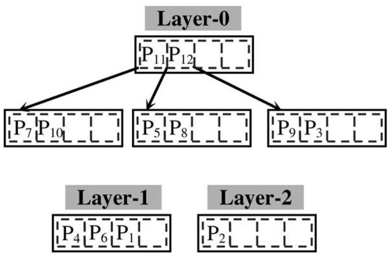

Fig. 3. The multi-layered 5-way B-trees from Fig. 2.

valid prefixes stored in a node, it is easy to know how many children this node has and thus any branch pointer (i.e., index to the aggregate array) can be computed by adding the base address to the offset of the branch pointer. To further reduce the memory consumption, a space efficient prefix format called *lite prefix format* is used and a segmentation table [20] is used for each layer to truncate common most significant bits of prefixes in each segment. All these proposed techniques will be described in details below.

3.1 Lite Prefix Format

Two commonly used formats to represent the prefixes of various lengths are *mask* format and *length* format. For example, prefix 101^{***} in 6-bit address space can be represented as 101000/111000 in mask format or 101000/3 in length format. In general, each prefix in the W-bit address space needs $2W$ bits and $W + \lceil log_2(W+1)\rceil$ bits in mask format and length format, respectively. W is 32 in IPv4 or 128 in IPv6. In this paper, we focus on design for IPv4 since the proposed architecture can be applied to IPv6 similarly.

The lite prefix format used in the proposed hardware architecture is similar to $(W+1)$-bit prefix representation proposed in [10]. The definition of lite prefix representation in the W-bit address space is as follows: The prefix of length i, $b_{W-1}b_{W-2}\ldots b_{W-i}{}^*$, is represented as $c_W \ldots c_0$, where $c_j = b_{j-1}$ for $j = W$ to $W-i+1$, $c_{W-i} = 0$, and $c_j = 1$ for $j = W-i-1$ to 0, (i.e., $b_{W-1}b_{W-2}\ldots b_{W-i}$ is followed by a zero and $W-i$ consecutive ones). Comparing two prefixes is as simple as comparing two $(W+1)$-bit numbers. However, searching a matched prefix against the IP address d needs some extra operations after we know $A \le d \le B$ for two consecutive prefixes A and B in the array of sorted prefixes in a layer. We have to check if d is contained in either A or B by using the masks of A and B described below.

```
Algorithm LayeredTrees_Search(d, Trees[], I)
{  // d is the destination address, I is the # of existing layers
01  for (i = 0; i < I; i++) {
02     x = Trees[i]; // search starts from Trees[i], the root of layer i
03     while (x ≠ NULL) {
04        Search node x for d;
05        if (x.keys[j] Encloses d) return x.NH[j];
          // when no match is found in node x
06        if (d < x.keys[0]) x = x.branch[0];
07        if (x.keys[k] < d < x.keys[k + 1]) // k∈ {0.. m – 3}
08              x = x.branch[k + 1];
09        if (x.keys[m – 2] < d) x = x.branch[m – 1];
10     } // end-while
11  } // end-for
12  return default_NH;
}
```

Fig. 4. LayeredTrees search.

For a prefix $c_W \ldots c_0$ in the lite prefix format, its mask denoted by $m_{w-1}m_{w-2}\ldots m_0$ can be easily recovered by the logical equations $m_i = \overline{\prod_{j=0}^{i} c_j}, i = 0 \cdots W-1$. For example, the 6-bit prefix 1011^{**} is converted into 1011011 in 7-bit lite prefix format and its mask is 111100. For the routing table consisting of 300 K prefixes, the memory required for storing the prefixes in a linear array based on the lite prefix format is 9.9 Mbits while mask format and length format need 19.2 and 11.4 Mbits, respectively.

3.2 LayeredTrees Search

Assume the B-tree node is of order m. Thus, every B-tree node stores at most $(m-1)$ prefixes and m branches.

To perform a search with a destination address d, layer 0 is examined first. If a layer-0 prefix matches d, i.e., it covers d, this prefix must be the only matching prefix at this layer since all the prefixes in the same layer are disjoint. It is possible that some other matching prefixes can be found in other different layers. However, they must be shorter than the matched prefix found in layer 0. In general, if a match is found in a layer $i-1$, there is no need to search the layer i or higher. If a match is not found in layer $i-1$, we have to continue the search operation in layer i. Fig. 4 shows the search algorithm in which I is the total number of layers. Line 4 checks if there exists any key (i.e., $x.keys[j]$) in node x that matches d by performing the match operations between d and all the keys in parallel that can be easily done by the hardware design. In lines 5-6, if a match is found, the procedure returns the next-hop information of the matched prefix. Otherwise, it determines which branch should be followed at the next step in lines 6-9. If no match is found in current layer, the search continues in the next layer.

3.3 LayeredTrees Insert

The LayeredTrees insertion algorithm is shown in Fig. 5. To insert a prefix p, the first layer (layer-0) is traversed first. Consider the insertion in layer i. If a prefix (say $x.keys[j]$) is found to be covered by p, the insertion process continues by inserting p in the next layer, as shown in line 6. If p is enclosed by the key (say $x.keys[j]$, $x.keys[j]$ should be replaced by p and then the insertion process continues by inserting $x.keys[j]$ in the next layer, as shown in lines 7-10. If no prefix covers p or is

```
Algorithm LayeredTrees_Insert(p, Trees[], I)
{ // p is the prefix to be inserted, I is the # of existing layers
01 for ($i = 0$; $i < I$; $i$++) {
02     $x$ = Trees[$i$];
03     while ($x \neq$ NULL) {
04         Search node $x$ for $p$;
05         if ($x$.keys[$j$] == $p$) return $I$; // $p$ already exists
06         if ($p$ Encloses $x$.keys[$j$] ) break;  // $p$ should be in higher layer
07         if ($x$.keys[$j$] Encloses $p$) {
08             temp = $x$.keys[$j$]; $x$.keys[$j$] = $p$; $p$ = temp;
09             break;
10         }
11         if ($x$ is leaf node) { BTree_Insert(Trees[$i$], $p$); return $I$;  }
12         if ($p < x$.keys[0]) $x = x$.branch[0];
13         if ($x$.keys[$k$] $< p < x$.keys[$k + 1$]) // $k \in \{0..\ m - 3\}$
14             $x = x$.branch[$k + 1$];
15         if ($x$.keys[$m - 2$] $< p$) $x = x$.branch[$m - 1$];
16     } // end-while
17 } // end-for
18 BTree_Insert(Trees[$I$+1], $p$); return ($I$+1);
}
```

Fig. 5. LayeredTrees insert.

covered by p, prefix p must be disjoint from all layer-i prefixes. Thus p is inserted into layer i, as shown in line 11. This insertion process is repeated by following the appropriate branches until p is inserted in any existing layer or a new layer must be created to hold p.

3.4 LayeredTrees Delete

The deletion algorithm is shown in Fig. 6. Deleting a prefix must follow the layer grouping constraint. Assume the prefix to be deleted, p, is found in layer i starting from layer 0. We have to make sure whether or not p can be deleted from layer i directly by the following steps. First, layer $i + 1$ is searched to see if there is a prefix q that covers p. If q does not exist (line 7) or q covers another prefix in layer i (lines 8-13), then p can be deleted directly from layer i and the deletion process is finished. The process of finding if prefix q in layer $i + 1$ covers not only p but also another prefix in layer i can be accomplished by simply determining if the left sibling (l) or right sibling (r) of p in layer i is also covered by q. If p is the only prefix in layer i covered by q, the layer grouping constraint will be violated if p is deleted directly. In this case, q should be lowered by one layer and put in layer i (line 14). The process of deleting p in layer i is then converted into that of deleting q in layer $i + 1$. If p encloses any prefix in layer i, p may exist in next higher layers and thus the deletion process continues in layer $i + 1$ (line 16). Finally, the process of determining which branch of the current B-tree node to follow is performed in lines 18-21.

3.5 Controlled B-Tree Building Algorithm

To improve the utilization of B-tree nodes, an algorithm named “Controlled B-tree Building Algorithm” (CBA) is proposed to increase the utilization of B-tree nodes. CBA inserts prefixes in an increasing order of their prefix values and splits at the position of the last key, instead of the middle key of the node when the B-tree node is full.

The prefixes of each layer are inserted into the corresponding B-tree one by one in the increasing order of their prefix values in lite prefix format. When a prefix is inserted into a full

```
Algorithm LayeredTrees_Delete(p, Trees[], I )
{ // p is the prefix to be deleted, I is the # of existing layers
01 for ($i = 0$ ; $i < I$ ; $i$++) {
02     $x$ = Trees[$i$];
03     while ($x \neq$ NULL) {
04         Search node $x$ for $p$;
05         if ($x$.keys[$j$] == $p$) {
06             find $q$ in layer-($i$+1) such that $q$ encloses $p$;
07             if ($q$ does not exist) {BTree_Delete(Trees[$i$], $p$); return;}
08             $l$ = Predecessor($x$); // left sibling of $x$.keys[$j$]
09             if ($l \neq$ NULL and $q$ Encloses $l$ ) {
10                 BTree_Delete(Trees[$i$], $p$); return;  }
11             $r$ = Successor( $x$ );//right neighbor of $x$.keys[$j$]
12             if ($r \neq$ NULL and $q$ Encloses $r$) {
13                 BTree_Delete(Trees[$i$], $p$); return;  }
14             $x$.keys[$j$] = $q$; $p = q$; break;
15         } // end-if in line #05
16         if ($p$ Encloses $x$.keys[$j$] )  break; // $p$ may be in next layer
17         if ($x$.keys[$j$] Encloses $p$ ) return; // $p$ does not exist
18         if ($p < x$.keys[0]) $x = x$.branch[0];
19         if ($x$.keys[$k$] $< p < x$.keys[$k + 1$]) // $k \in \{0..\ m - 3\}$
20             $x = x$.branch[$k + 1$];
21         if ($x$.keys[$m - 2$] $< p$) $x = x$.branch[$m - 1$];
22     } // end-while
23 } // end-for
}
```

Fig. 6. LayeredTrees delete.

m-way B-tree node, the node is split at the position the last key, i.e., one contains $m - 2$ keys and the other contains only one key. According to our analysis, CBA can improve the utilization of B-tree node to 90 percent.

3.6 Aggregate Array and Segmentation Table

In order to store entire routing table in the on-chip memory, the aggregate array of B-tree nodes is used to store all m child nodes of a B-tree node. Thus, only one pointer called *base address* is needed in a B-tree node, instead of m branch pointers. When a B-tree node has k children, it must have $k - 1$ valid prefixes whose values must not be $11\ldots1$ based the lite prefix format. The aggregate array of B-tree nodes with k children is of size $k * S$ bytes, where S is the node size. The ith branch pointer of the B-tree node with base address B points to the address of $B + i * S$ for $i = 0$ to $k - 1$. The base address of a leaf node is set to zero to indicate that no branch exists.

Using a segmentation table [20] is always the simplest way to partition a tree into several smaller ones without complicating the search operations. Segmentation table reduces the tree height. Consider the segmentation table of s bits that partitions the prefixes into 2^s segments. If the length of prefix P_i to be inserted is $t \leq s$, P_i will be duplicated into 2^{s-t} segments. Since routing tables usually contain prefixes of length larger than or equal to 8, an 8-bit segmentation table that results no prefix duplication is used in our design.

Fig. 7 illustrates the B-trees in LayeredTrees using a 2-bit segmentation table based on the one without segmentation table in Fig. 3. All the segments record the *default_NH* as their default next-hops except that segment 2 has P_1 as its default next-hop because P_1 is of length 2. Segment 0 contains three pointers pointing to the root nodes of the three B-trees corresponding to layers 0, 1, and 2. The pointer of layer 0 of segment 0 points to the node A of the aggregate array. The layer 0 in segment 0 is a two-level B-tree where the root is node A in level 0

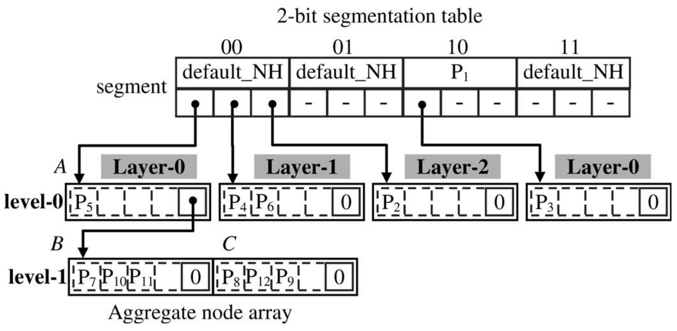

Fig. 7. A 5-Way B-tree built by CBA from Fig. 3.

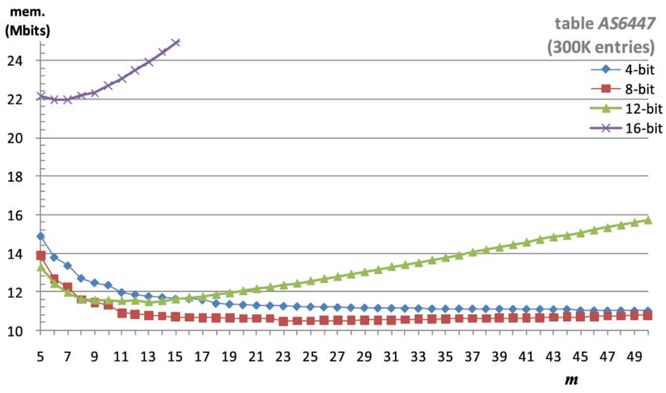

Fig. 8. Memory consumption.

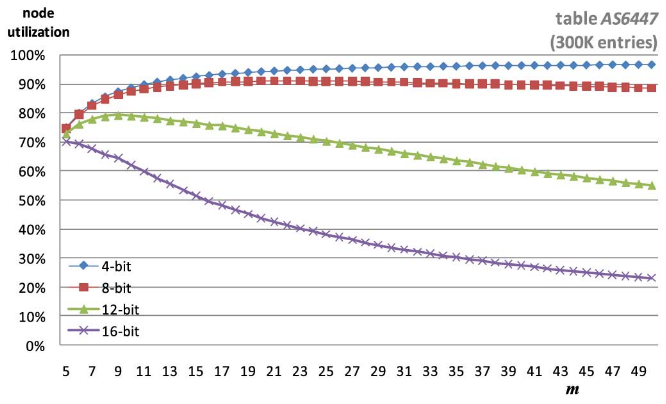

Fig. 9. Node utilization.

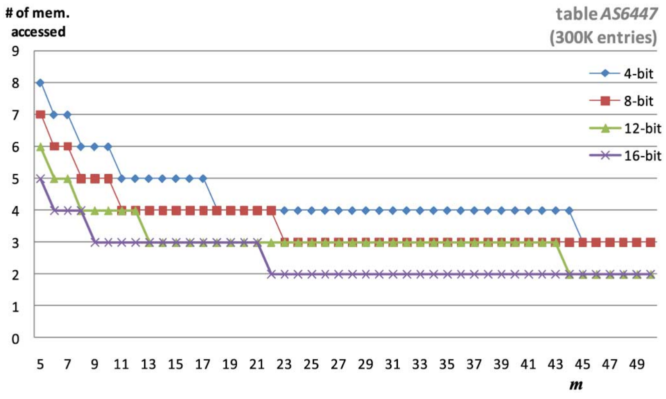

Fig. 10. Average number of memory Accesses.

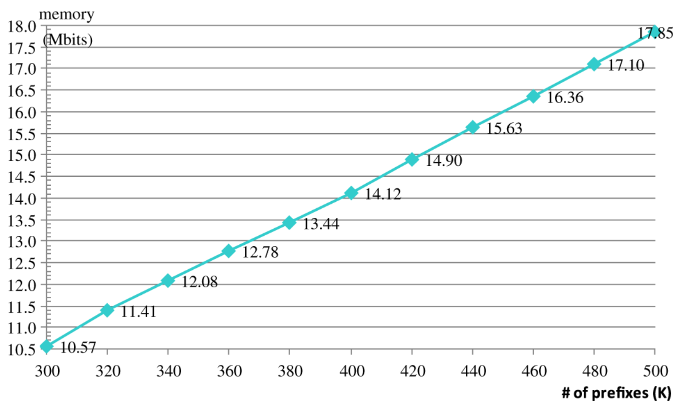

Fig. 11. Memory consumption for large tables.

containing a key P_5 and a base pointer pointing to node B of the aggregate array. Node B and its succeeding node C in the aggregate array correspond to the two branches of node A. All other B-trees corresponding to layers 1 and 2 of segment 0 and layer 0 of segment 2 contain only root nodes.

3.7 Varying Segmentation Table and Node Sizes

The segmentation table size denoted by s and B-tree node size denoted by m are two main factors effecting the memory consumption of the proposed LayeredTrees. Subsequently, we shall show the analysis results that can be used to select the best combination of s and m in terms of memory consumption, where $s \in \{4, 8, 12, 16\}$ in terms of number of bits and $m \in \{5\ to\ 50\}$ in terms of orders in B-tree nodes. We use the routing table AS6447 as an example. Fig. 8 shows that for the same node size, LayeredTrees with *4*-bit and *8*-bit segmentation table performs better because the memory consumption for both *12*-bit and *16*-bit segmentation tables increases as m increases. Fig. 9 shows the average node utilization, where node utilization is defined to the ratio of the number of valid keys stored in the node and $m - 1$. As we can see that 4-bit and 8-bit segmentation tables have higher utilization than 12-bit and 16-bit segmentation tables. Average number of memory accesses for each lookup is usually used to estimate the search speed of a IP lookup scheme. As shown in Fig. 10, the average number of memory accesses decreases when m or s increases. According to the above analysis, $(s, m) = (8, 24)$ is the best combination for LayeredTrees.

To evaluate the memory consumption of LayeredTrees for larger tables, we generate large synthesized tables that follow the same prefix length distribution of AS6447. In Fig. 11, LayeredTrees is scalable to a half million of prefixes with the on-chip memory of 18 Mbits.

4 Proposed Pipelined Architecture

In this section, two pipelined architectures are proposed for LayeredTrees to improve the system throughput. Firstly, a 5-stage pipeline called *leveled search engine* (*LSE*) is proposed to perform the search operations in LayeredTrees, as shown in Fig. 13. LSE searches the B-trees of LayeredTrees in a level-by-level fashion from layer 0 to layer 6. LayeredTrees terminates the search process as soon as a match is found in any level of

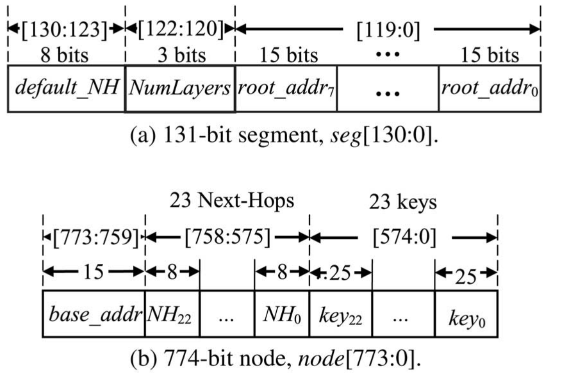

Fig. 12. Segment and 24-way B-tree node formats.

any layer. In other words, all the levels of all layers in LayeredTrees are searched one by one by using the 5-stage search engine. To take advantage of the parallel nature of the proposed LayeredTrees, a parallel architecture combined with multiple modified LSE's is then designed for further improving the throughput. Based on this parallel architecture, the worst case search time is reduced to be the delay to search the level with maximum amount of memory.

In order to describe the proposed architecture accurately and concisely, some notations and definitions are needed as follows.

- n: the size of the prefixes in bits.
- m: the order of the B-trees.
- W: the size of the IP address space in bits and W is 32/128 for IPv4/IPv6.
- s: the number of bits used in the segmentation table, i.e., $s = W - n$.
- *seg*: the segment entry of the s-bit segmentation table.
- $seg.root\_addr_i$: the root node address of the i^{th} layer of a segment, where $0 \le i \le 7$.
- *seg.NumLayers*: the number of layers in the segment.
- *seg.Default_NH*: the default next-hop in the segment.
- *node*: the B-tree node.
- *node.base_addr*: the base address of an aggregate array of B-tree nodes pointed to by their parent node.
- $node.key_i$: the i^{th} prefix in the node, where $0 \le i \le m - 2$.
- $node.NH_i$: the next-hop associated with $node.key_i$.

We choose $m = 24$ and $s = 8$ because there is no prefix of length less than 8 in the real-life IPv4 routing tables and $m = 24$ makes the height of B-trees in LayeredTrees no more than three levels. The 8-bit segmentation table needs 256 131-bit segments shown in Fig. 12a and the 774-bit 24-way B-tree node is also shown in Fig. 12b.

4.1 Leveled Search Engine (LSE)

LSE searches one B-tree node in a level at a time by going through all five stages (called a *round*) of the proposed pipelined architecture. Stage 1 determines the level and layer numbers of the current round. Initially, *start* signal (also called reset signal) is enabled to input a new IP address as the search key and then the values of *layer.no* (current layer number) and *level.no* (current level number) are reset. The finish signal returned from stage 5 is also used to select a new IP address as the search key for starting a new round. If neither *start* signal nor *finish* signal is enabled, the returned values of *layer.no* and *level.no* from previous round are used in the next round. Normally, if the levels of the current layer are not exhausted in a round, *level.no* is increased by one and *layer.no* is not changed. Otherwise, if a leaf node of the current layer is reached, *layer.no* is increased by 1 and *level.no* is reset to 0.

Stage 2 looks up the segmentation table by using the most significant 8 bits of the IP address and the current 3-bit layer number to retrieve the root node address of the B-tree in current layer. Each segment of the 8-bit segmentation table is 131 bits as shown in Fig. 12a. The root node address of the current layer is used as the address of the current node when the level number is zero. Otherwise, the node address returned from the previous round is used as the address of the current node. The default next-hop number (*seg.NH*) and the number of layers (*seg.NumLayers*) of the corresponding segment shown as seg[130:120] in stage 2 will be passed through the pipeline to stage 5. The default next-hop (i.e., *default_NH*) is needed when no match is found in all the layers.

All the B-tree nodes are stored in the memory of stage 3. A 774-bit node is used in this paper because a 24-way B-tree node needs 23 25-bit keys and 23 8-bit next-hops. Since an aggregate array of nodes is used to avoid storing the branch pointers physically, we need a base address of the aggregate array in each node. The base addresses of the leaf nodes are set to NULL which is defined to be $\overbrace{1 \cdots 1}^{15}$ in this paper. Thus, a AND gate is sufficient to check whether a node is a leaf node. The address of the current node is used to read the 774-bit node to be processed in stage 4. For the largest routing table used in our experiments, we need no more than 2^{15} 24-way B-tree nodes. Therefore, the base pointer is set to 15 bits wide. The address of the node to be processed in the next round is calculated in stage 5, by using the branch number and the base address of the current node.

The most complicated part of the proposed LSE is in stage 4 which contains two components, *IP/Keys Matching unit* and *Branch Detection unit*. These two components are processed in parallel. The IP/Keys Matching unit checks if the input IP address matches one of the keys stored in the node. The result returned from the Branch Detection unit is not needed in the following two cases. First, when a matched key is found in the current node by the IP/Keys Matching unit, the search for the current input IP is completed in the current round and so no next round is needed. Second, when the base address of the current node is zero, i.e., the current node is the leaf node of the current layer, the search will continue in the next layer or the default NH will be the final result if the current layer is the last layer. If these two cases do not happen, the result returned from the Branch Detection unit is used in stage 5 to compute the address of the node in the next B-tree level. Notice that the keys are the prefixes in the lite prefix format. For n-bit prefixes, each key in lite prefix format is of size $n + 1$ bits. Although the lite prefix has the advantage of low memory consumption compared with other prefix formats, the match operation between the input IP and the lite prefix can not be made easily and efficiently. Therefore, we choose to use the mask format to perform the match operations. However, we have to compute the subnet mask from the lite prefix first. Suppose that a node only stores $h < m - 1$ keys in the increasing order of their key values (i.e., $key_0, \ldots, key_{h-1}$), where $key_k > key_{k-1}$

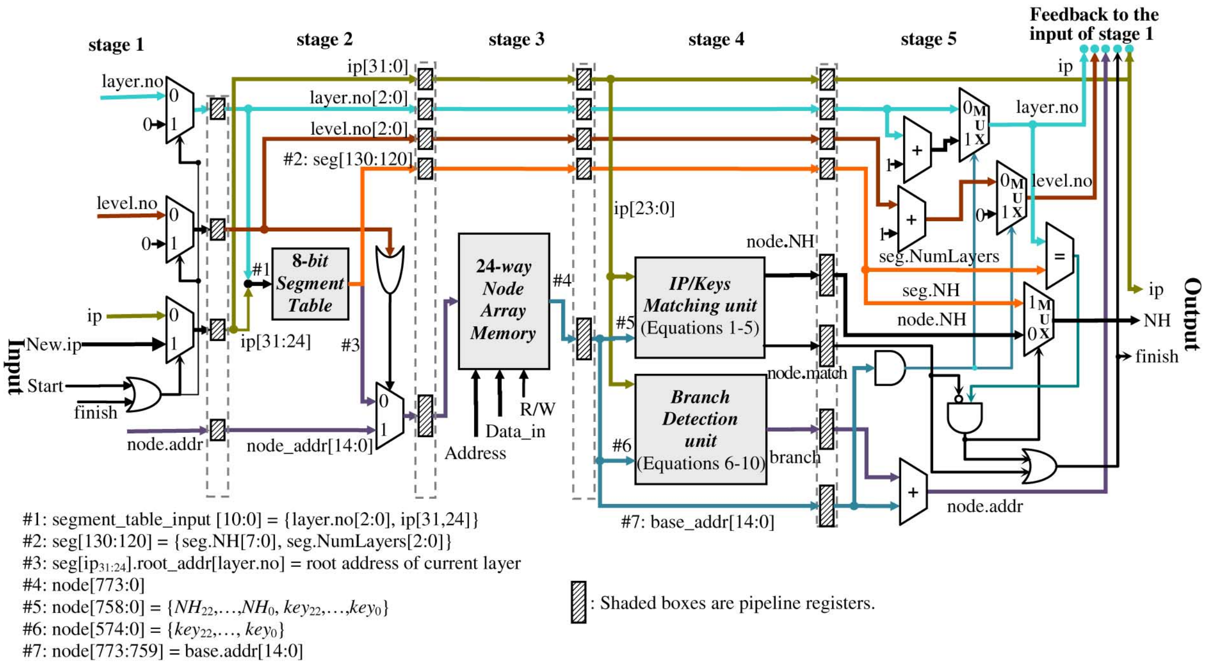
stage 1
stage 2
stage 3
stage 4
stage 5
Feedback to the input of stage 1
Input
layer.no
level.no
ip
New.ip
Start
finish
node.addr
ip[31:0]
layer.no[2:0]
level.no[2:0]
#2: seg[130:120]
8-bit Segment Table
#1
ip[31:24]
#3
node_addr[14:0]
24-way Node Array Memory
#4
Address
Data_in
R/W
ip[23:0]
node.NH
IP/Keys Matching unit (Equations 1-5)
#5
Branch Detection unit (Equations 6-10)
#6
node.match
branch
#7: base_addr[14:0]
seg.NumLayers
seg.NH
node.NH
layer.no
level.no
node.addr
Output
ip
NH
finish
#1: segment_table_input [10:0] = {layer.no[2:0], ip[31,24]}
#2: seg[130:120] = {seg.NH[7:0], seg.NumLayers[2:0]}
#3: seg[ip31:24].root_addr[layer.no] = root address of current layer
#4: node[773:0]
#5: node[758:0] = {NH22,…,NH0, key22,…,key0}
#6: node[574:0] = {key22,…, key0}
#7: node[773:759] = base.addr[14:0]
: Shaded boxes are pipeline registers.

Fig. 13. The 5-Stage Leveled Search Engine.

for $k = 1$ to $h - 1$. Each of the $m - h - 1$ unused slots stores the $NULL$ value which is defined to be $\overbrace{1\cdots1}^{n+1}(n+1$ ones) in this paper. $NULL$ is the only $(n+1)$-bit number that is not used by the lite prefix format. The detailed logic design of the IP/Keys Matching unit is shown in the following five equations.

$$NULL_k = \prod_{j=0}^{n} \overrightarrow{key_k}[j], \tag{1}$$

$$\overrightarrow{M_k}[i] = \overline{\prod_{j=0}^{i} \overrightarrow{key_k}[j]}, i = 0 \cdots n-1, \tag{2}$$

$$match_k = \overline{NULL_k} \cdot \overline{\sum_{i=0}^{n-1} \left(\overrightarrow{ip}[i] \oplus \overrightarrow{key_k}[i+1] \cdot \overrightarrow{M_k}[i] \right)}, \tag{3}$$

$$\overrightarrow{node.NH}[j] = \sum_{k=0}^{m-2} \left(\overrightarrow{NH_k}[j] \cdot match_k \right), j = 0 \cdots 7, \tag{4}$$

$$node.match = \sum_{k=0}^{m-2} match_k. \tag{5}$$

Equation (1) computes a flag called $NULL_k$ for the k^{th} key (i.e., the n-bit vector $\overrightarrow{key_k}$) in the lite prefix format. If the k^{th} key is NULL, $NULL_k$ is set to 1. Otherwise, $NULL_k$ is set to 0. The mask $\overrightarrow{M_k}$ of the kth key is computed in equation (2). From (2), the mask of the NULL key is $\overbrace{0\cdots0}^{n}$. The input IP address denoted by the n-bit vector $\overrightarrow{ip}$ is matched against mask $\overrightarrow{M_k}$ to get the matching result $match_k$ in equation (3). The right-hand side of the AND operation in equation (3) is the conventional match operation between a IP address and a prefix, which will be always true if the mask is $\overbrace{0\cdots0}^{n}$. Thus, to force the matching result $match_k$ between an IP address and the mask of a NULL key to be always false, the left-hand side $\overline{NULL_k}$ of the AND operation in equation (3) is needed. Equation (4) computes the next-hop number corresponding to the matched key and equation (5) sets output signal *node.match* to one if when one of $m-1$ keys matches input IP. Two results output from the IP/Keys Matching unit, *node.match* (1 bit) and *node.NH* (8 bits), are needed in the next stage.

The design of IP/Keys Matching unit in equations (1)–(5) can be simplified as follows. As we know that n is 32 for IPv4 and $\overbrace{1\cdots1}^{n}$ (i.e., 255.255.255.255) represents the limited broadcast address that will not be forwarded by the router. Thus, we can assume that address $\overbrace{1\cdots1}^{n}$ does not appear in any unicast packet. We can use $\overbrace{1\cdots1}^{n}0$ (i.e., the lite prefix of the IP address 255.255.255.255) as the NULL value. The mask of $\overbrace{1\cdots1}^{n}0$ is $\overbrace{1\cdots1}^{n}$ and thus no IP address is able to match the unused slots. As a result, the flag $NULL_k$ in equations (1) and (3) can be removed completely.

When no match is found in a non-leaf node, we have to find a branch that we can traverse to reach a node in the next level for continuing the search operation. Equations (6)–(10) show the detailed design of the Branch Detection unit.

$$\overrightarrow{g_k}[0] = \overrightarrow{key_k}[0], \overrightarrow{g_k}[i] = \overrightarrow{key_k}[i] \cdot \overline{\overrightarrow{ip}[i-1]}, i = 1 \cdots n, \tag{6}$$

$$\overrightarrow{e_k}[0] = \overline{\overrightarrow{key_k}[0]}, \overrightarrow{e_k}[i] = \overline{\overrightarrow{key_k}[i] \oplus \overrightarrow{ip}[i-1]}, i = 1 \cdots n, \tag{7}$$

$$G_k = \sum_{i=0}^{n} \left(\overrightarrow{g_k}[i] \cdot \prod_{j=i+1}^{n} \overrightarrow{e_k}[j] \right), \tag{8}$$

$$\overrightarrow{d}[0] = \overline{G_0},\ \overrightarrow{d}[m-1] = G_{m-2}, \text{and}$$
$$\overrightarrow{d}[k] = G_{k-1} \oplus G_k, k = 1 \ldots m-2, \tag{9}$$

$$\overrightarrow{branch}[l] = \sum_{i=1}^{\lceil m/2^{l+1} \rceil} \left(\sum_{j=i\times 2^{l+1}-2^l}^{min(i\times 2^{l+1}-1, m-1)} \overrightarrow{d}[j] \right). \tag{10}$$

The first three equations (6)-(8) determine if key_k is larger than the input IP ($\overrightarrow{ip}$) and stores the result in G_k. All G_k's for $k = 0$ to $m-2$ form an $(m-1)$-bit vector $\overrightarrow{G}$. Equation (9) converts $\overrightarrow{G}$ to an m-bit vector $\overrightarrow{d}$ such that only one bit in $\overrightarrow{d}$ is set and all other bits are unset. The last equation (10) computes the branch number, i.e., the bit position of the only set bit in $\overrightarrow{d}$ and stores it in the $\lceil \log_{2m} \rceil$-bit vector, $\overrightarrow{branch}$.

Stage 5 decides whether the search operation for the current IP address should be finished at the current level. The *finish* signal is set when the *node.match* signal from the IP/Keys Matching unit is true or no match is found after searching all the layers in the corresponding segment. Computing *finish* signal involves a 3-bit comparator, an AND gate, and an OR gate. The comparator compares the number of layers stored in the segment and the current layer number. The output of the AND gate is used as the selector of a multiplexor to select either *node.NH* or *seg.NH* (i.e., the default port of the segment). Two adders are used to increment the current level number (*level.no*) and current layer number (*layer.no*) by one. The operations for computing these two addresses are done by an AND gate, two adders, and two multiplexors. A final adder is used to compute the node address pointed to by the branch computed in stage 4.

One special case that must be considered is when a segment contains no layers. We call this kind of segments *empty* segments. Since all the search operations must go through the pipeline at least one round, we restructure the empty segments as the ones containing one layer with only one level (i.e., one root node). Thus, the number of layers in the empty segment becomes one. Also, the only node in empty layers contains all the NULL keys defined above and a NULL base address. As a result, the proposed LSE works for empty segments.

4.2 Multiple Parallel Leveled Search Engines

As per LSE described above, the search goes through the 5-stage pipeline at most r_i times in r_i-level B-tree of layer i. In the worst case, all the layers must be searched. Thus, $r_{\max} = \sum_{i=0}^{S-1} r_i$ cycles are needed to complete a search for the routing table that needs S layers in the proposed Layered-Trees. Take the real-life routing table AS6447 2009-7 as an example. The proposed 24-way LayeredTrees with an 8-bit segmentation table needs seven layers as shown in Table 1 and r_{max} is 13. As we have shown in the proposed LSE, most of the search operations will be completed as early as in the first layer. Thus, in the best case, the search can be completed in the first round, i.e., by going through the 5-stage pipeline one to three times. As a result, the average number of cycles (r_{avg}) needed is much less than r_{max}. From our experimental results, r_{avg} is 3.02 for table AS6447 2009-7. Although r_{avg} is small compared to r_{max}, the proposed LSE exhibits two disadvantages. The first disadvantage is the long worst-case search delay which may not be acceptable for high-performance routers. The second disadvantage is the large variation of search delays for different packets that may result in the out-of-order packet delivery. Therefore, we will propose a parallel design that uses multiple LSE's running in parallel to solve these two disadvantages as follows.

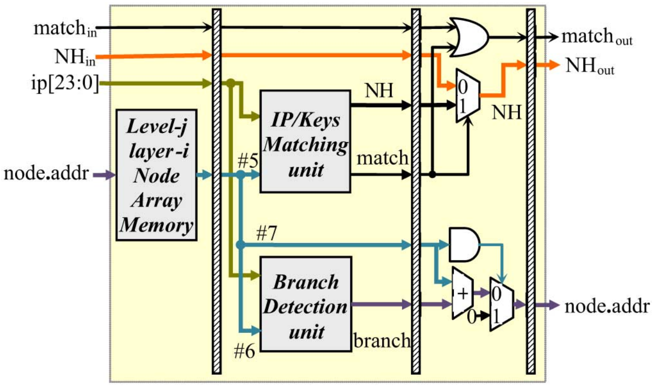

Fig. 14. The modified 3-stage search engine, $\mathrm{SE}_{i,j}$, where signals #5, #6, and #7 are defined in Fig. 13.

The proposed parallel design trades the hardware cost in terms of slices in FPGA devices for higher packet throughput and a constant packet delivery time. For layer i consisting of r_i levels, the proposed parallel design concatenates r_i copies of a new 3-stage pipeline modified according to stages 3-5 of the proposed LSE. The modified 3-stage pipeline shown in Fig. 14 is simpler because the signals *layer.no* and *level.no* are not needed. For the routing table consisting of $r_{\max} = \sum_{i=0}^{S-1} r_i$ levels, r_{max} copies of the modified 3-stage pipelines are required. In order to reduce the amount of memory required and improve the search speed, four optimization techniques are used: the *simpleone-level prefix push* scheme, *routing table split* scheme, *B-tree order varying* scheme, and *stage 4 clock rate improving* scheme.

The simple *one-level prefix push* scheme reduces the number of layers required and thus the hardware cost. Similar to the traditional prefix push algorithm in the binary trie, it only pushes the prefixes down one level if at least one of their child prefixes is valid prefix. For example, prefix P_2 in Fig. 2 can be pushed to its right child without changing the search outcomes of the routing table. However, prefixes P_1, P_4 and P_6 will not be pushed down because both of their child nodes are not valid. Based on the proposed one-level push, the number of prefixes in the routing table after push operations will not be increased. The main advantage of this push scheme is that the layer number assigned to some prefixes will be changed from high numbers to lower ones. For example, the layer number of P_2 will be changed from three to two after the 1-level push operations and the number of prefixes remains to be 12. In fact, the number of prefixes may be reduced because the prefixes whose both child nodes are valid can be deleted directly without affecting the search results.

The *routing table split scheme* is proposed to reduce the number of bits needed for the prefixes of lengths 9 to 24 stored in B-tree nodes. The original routing table is split into two groups: the *large* and *small* groups consisting of the prefixes of lengths 9 to 24 and lengths 25 to 32, respectively. The large group accounts for at least 98% of the prefixes in the routing

TABLE 2
Prefix Enclosure Analysis After Push Operations

Routing Table		AS6447 (2005-4)		AS6447 (2009-7)		AS6447 (2011-5)	
size		163,535		301,552		369,394	
length		9-24	25-32	9-24	25-32	9-24	25-32
Layer	0	7,071	3,488	271,548	5,736	333,034	5579
	1	496	67	17,081	28	20,884	21
	2	54		1,488		1,859	
	3	6	0	88	0	111	0
	4	0		2		5	

table. For the large group, only 17-bit keys are needed, instead of 25 bits. The memory consumption for prefixes in the small group remains the same. The prefix enclosure analysis after push operations is shown in Table 2. We can see that the total number of layers for both large and small groups is 7 while the number of layers in the original table is 8 for table AS6447 2011-5.

The third *B-tree order varying* optimization scheme uses as small a B-tree order as possible for the B-tree nodes in each layer. As obtained by our experiments, the order of the 3-level B-trees for layer 0 is 22. Since other layers contain much less prefixes than layer 0, we can shrink the required memory by reducing the B-tree order of these layers as long as the number of levels in the B-trees is not greater than 3. Take table AS6447 2009-7 as an example. For the large group of prefixes of lengths 8-24, five layers are needed and their B-tree orders are 22, 10, 4, 2, and 2. Similarly, the B-tree orders of the two layers required for the small group of prefixes of lengths 25-32 are 9 and 2.

The final *stage 4 clock rate improving* optimization scheme focuses on increasing the clock rate of the bottleneck stage of the proposed pipelines. As described above, the bottleneck of the proposed pipelines is the IP/Keys Matching unit in stage 4. Therefore, we break stage 4 (i.e., equations 1 to 5) into three sub-stages computing equations 1-2, 3, and 4-5, respectively. Since Branch Detection unit runs in parallel with the IP/Keys Matching unit, it is also split into three sub-stages computing equations 6-8, 9, and 10 respectively. Notice that the original design of the proposed LSE is 5-stage pipeline architecture and the modified LSE remains to be a 5-stage pipeline because stages 1-2 of the original LES are removed and two additional stages are introduced in stage 2 of the modified 3-stage pipeline.

Fig. 15 illustrates the parallel design for table AS6447 2011-5 by using the proposed table split scheme. $SEL_{i,j}$ and $SES_{i,j}$ denote the modified 3-stage pipelines for level j of layer i in the large and small group, respectively. In $SEL_{i,j}$, IP [23, 8] is the input in the 3-stage pipeline, instead of IP[23, 0] in the original LSE. Because the maximum number of levels needed among all layers is 3, the total number of pipeline stages needed in the parallel design with an 8-bit segmentation table and 24-way nodes is 17 including the stage for input IP address and 8-bit segmentation table and the stage for priority encoder. Also, the default port of each segment denoted by seg.NH is only input into the last pipeline, e.g., $SEL_{4,0}$ in Fig. 15. So, at least one match can be found in the last pipeline. The priority encoder will output the first match from top to bottom.

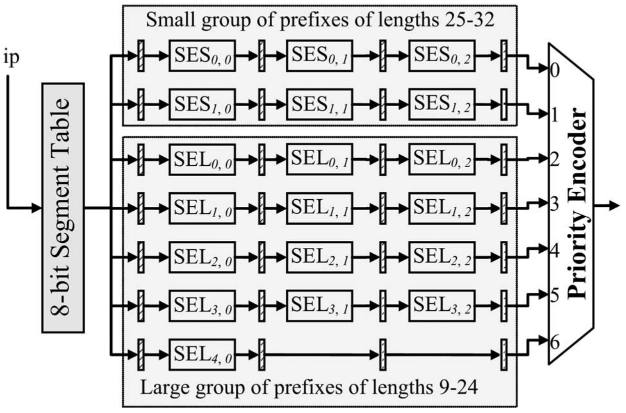

Fig. 15. Parallel Leveled Search Engines.

5 PERFORMANCE

To conduct the performance evaluation, the pipeline architecture of the proposed 24-way B-trees with an 8-bit segmentation table is implemented by using Xilinx ISE 12.2 with Virtex-6 XC6VSX315T FPGA chip [42] that contains 49,200 slices and a Block RAM of size 25,344 Kb. All packets are assumed to be 40 bytes, the minimal Ethernet packet size. Let C denote the pipeline clock rate in nano-second (ns) and R denote the number of nodes accessed per search. For the proposed LSE, the throughput in terms of million packets per second (Mpps) is $1000/(C \times \mathrm{R})$, and the throughput in terms of billion bits per second (Gbps) is $320/(C \times R)$. For the parallel LSE, the throughput is $1000/C$ Mpps or $320/C$ Gbps.

The performance results are obtained from the simulations on Xilinx ISE tool. The correctness of the proposed design is verified by the test bench. The component tested in the simulations is the forwarding engine of the router based on the proposed LSE and parallel LSE. Other components required in the router are not considered in the simulations. When the other components of the router are as fast as the proposed design, we expect that an actual implementation using parallel LSE can achieve very high performance for both IPv4 and IPv6. Table 3 shows the performance of the proposed LSE and parallel LSE compared with other existing hardware implementations, DMST [11], Ring [1], CAMP [23], OLP [20], BiOLP [24], BPFL [13], POLP [13], [22], Bit-Shuffled Trie (BSTrie) [34], prefix partitioning based on Binary Search Tree (BST) and 2-3-Tree [28], and Distance Bounded Path Compressed trie (DBPC) [27]. Since we are not able to implement all these existing schemes based on the same routing tables and FPGA devices, we list the reported results that use the tables of various sizes published in their original papers. FlashLook [3] and FlashTrie [4] that can also achieve a throughput of 100 Gbps are not included in the comparisons because they use DRAM instead of on-chip memory in their main data structures of IP lookups. The endpoint-based OnChip [40] design is also not included in the comparisons because no results of FPGA implementation were reported in the original paper.

LSE can achieve the throughputs of 33.6 Gbps and 7.8 Gbps in the average case and the worst case, respectively. The bottleneck of the LSE pipeline occurs in stage 2. Since the parallel LSE simplifies stage 2, its pipeline bottleneck moves to the second sub-stage of stage 4. As a result, the clock rate of

TABLE 3
Performance Comparisons for IPv4 Tables

Scheme	FPGA Device	Configurable logic (slice)	Memory (Mbits)	# of prefixes	ME ratio (bits/prefix)	Clock rate (ns)	# of nodes accessed per search		Throughput			
									Mpps		Gbps	
							avg	worst	avg	worst	avg	worst
LSE	Virtex-6 XC6VSX315T	1,402 (2.8%)	10.9	301,552	37.0	3.157	3.02	13	104.9	24.4	33.6	7.8
Parallel LSE		7,408 (15%)	10.8		36.8	2.649	1		377.5		120.8	
DMST [11]	Virtex-5 VLX330T Original	463 (1%)	11.68	163,574	74.9	5.0	4.68	6	42.7	33.3	21.9	17.1
	Extended	2,778 (6%)			74.9	5.0	1.0		200		102	
Ring [1]	Virtex-2P XC2VP70	408 (1%)	3.49	30,000	116.4	4.0	2.0		125		40	
CAMP [23]		528 (2%)	3.89		129.6		1.3		200		64	
OLP [20]		362 (1%)	3.46		115.2		1.0		250		80	
BiOLP[(a)(c)(d)][24]	Virtex-4 FX140	1,785 (3%)	9.54	83,662	114.0	3.069	1.0		325		104	
BPFL[(a)] [13]	Stratix II EP2S180F1020C5	27.9K LE [(b)] (19%)	5.6	309,000	18.1	10.35	1.0		96.6		30.9	
POLP[(a)] [13]		6.6K LE [(b)] (5%)	9.08		29.4	9.29	1.0		107.6		34.4	
pDST[(d)] [43]	Virtex-5 LX330	12.8k LUTs(6.3%)	7.112	96,000	75.86	7.4	1.0		242		77.4	
BSTrie[(a)] [34]	Virtex-5 XC5VSX240T	749 (2%)	7.38	321,000	23.0	5.7	1.0		175		56	
DBPC [27]	Virtex-6 XC6VSX475T	3,524 (5%)	26.58	330,000	80.55	4.286	1.0		233		74.56	
BST [28]	Virtex-6 XC6VSX475T	9,054 (12%)	14.13	330,000	42.82	4.875	1.0		205		65.6	
2-3-Tree [28]		10,453 (14%)	20.21	330,000	61.24	4.986	1.0		200		64	

(a) The nexthops are not stored in on-chip memory.
(b) Instead of slices, Stratix LEs are used.
(c) It is the non-cache based BiOLP.
(d) dual-ported block RAM is used.

parallel LSE is improved from 3.058 ns to 2.649 ns. The final throughput that can be achieved by the proposed parallel LSE is 120.8 Gbps. Other than the proposed designs, BiOLP [24] performs the best.

The FPGA devices have a memory usage constraint: the allocated memory unit is restricted to a block of $1024s \times 18w$ bits (i.e., 1024 s entries of $18w$ bits wide), where s and w are integers. For example, in Fig. 13, the stage 2's 8-bit segmentation table whose detailed structure is shown in Fig. 12a needs a memory block of 1024×144 bits and the 24-way node array of N nodes in stage 3 as shown in Fig. 12b takes a memory block of $1024\lceil N/1024 \rceil \times 774$ bits. The memory utilization of parallel LSE is lower than LSE because parallel LSE needs many smaller memory blocks for parallel and pipelined processing as shown in Fig. 15. However, our design that divides prefixes into small and large groups amends this memory usage inefficiency in parallel LSE and thus the memory requirement of parallel LSE becomes very close to that of LSE. Because the routing tables of various sizes are used in the experiments of the existing designs, we use a metric called *ME ratio* [40] to compare the memory usages. ME ratio is defined to be the average amount of memory in bits needed per prefix. If M is total amount of memory in bits required to store the entire routing table in on-chip memory for a design and N is the number of prefixes of the routing table, ME ratio is M/N. BPFL, POLP, and BSTrie have lower ME than the proposed schemes. One of the reasons is that all these three designs do not store the next-hop numbers in their data structure stored in on-chip memory. Therefore, an extra step is needed to access the next-hop numbers after the search is completed in the on-chip memory. Notice that since we just store each prefix once in LSE and Parallel LSE, the memory requirement for the nexthops is $8N \text{ bits} = 301,552 \times 8 = 2.3$ Mbits. Thus, if nexthops are not stored in LSE and Parallel LSE, the memory requirement for LSE and Parallel LSE will be reduced to 8.6 and 8.5 Mbits, respectively.

The performance of updates is shown in Table 4. The 10-minute real-life update trace files for routing tables with the prefix "rrc" are retrieved from RIPE website [35]. Because most update operations are to change the next-hop information for a prefix merely, the average number of B-tree nodes that node need to be modified per update is 1.02.

The update operations have minimal influences on the search performance. The read operation of the on-chip SRAM at the search engine is executed in every cycle. We use the precedence model in our design. The write operation

TABLE 4
The Update Performance for AS6447 2009-07

Update traces	# of entries	# of updates	avg # of nodes modified / update
rrc00	317,670	42,542	1.03
rrc01	308,943	51,591	1.01
rrc03	307,598	47,837	1.01
rrc04	318,309	14,181	1.03
rrc05	308,285	29,934	1.01
rrc06	307,275	3,113	1.04
rrc07	307,802	28,595	1.02
rrc10	306,690	10,871	1.04
rrc11	308,846	42,662	1.01
rrc12	305,608	16,115	1.02
rrc13	319,720	34,792	1.01
rrc14	310,403	18,826	1.03
rrc15	307,538	13,396	1.01
rrc16	313,948	4,414	1.02

TABLE 5
Performance Comparisons for IPv6 Tables

Scheme	FPGA Device	Configurable logic (slice)	Memory (Mbits)	# of prefixes	ME ratio (bits/prefix)	Clock rate (ns)	# of nodes accessed per search		Throughput			
									Mpps		Gbps	
							avg	worst	avg	worst	avg	worst
LSE	Virtex-6 XC6VSX315T	969 (1%)	0.94	7,049	136.0	3.058	3.03	9	107.9	36.3	34.5	11.6
Parallel LSE		2,9198 (5%)	1.58		230.1	2.416	1		413.9		132.5	

has precedence over the read operation when a write operation is needed for an update with a read operation in some cycle. Therefore, instead of retrieving correct data for the requested search, the read operation obtains the data which is written by the update. It makes retrieved next-hop information incorrect and causes the packet to be dropped or forwarded to a wrong destination. In TCP/IP protocol, the problem of packet lost would be detected and the lost packet is retransmitted again.

Consider the rate of 1000 updates per second. There will be 1,020 node modifications that will cause at most 1,020 packets being dropped or forwarded incorrectly in every second. Compared to the 377.5 million packets being forwarded per second by the proposed parallel LSE, 1,020 (the number of packets being dropped or forwarded incorrectly) is very small and can be ignored. Thus, the search speed remains the same when updates are taken into consideration.

Notice that one way to avoid the small number of the corrupted lookups stated above is to update the memory in the pipeline by inserting write bubbles as proposed in [5]. The B-tree nodes in all stages that need to be modified due to a prefix update are computed offline. The change of a B-tree node in a stage is enabled only after the corresponding write bubble arrives at that specific stage. Since one route update may cause multiple B-tree nodes in a level (stage) to be modified, many write bubbles need to be inserted in a row. One disadvantage of using write bubbles for updates is as follows. The complexity of the pipeline architecture will increase because many control lines need to be added into the design and normal lookup operations and write bubbles need to be differentiated in each stage. As a result, the overall hardware cost will be increased and the clock rate will be degraded. Therefore, we go for the simple approach instead of using write bubbles to perform the update operations.

We also perform the implementation for IPv6 tables. Table 5 shows the performance results for a real-world routing table consisting of 7,049 entries which is the largest IPv6 table currently available [7]. For LSE, the achieved throughputs are 34.5 and 11.6 Gbps for the average and worst cases, respectively. For the parallel LSE, the achieved throughput is 132.5 Gbps. This throughput is a little better than that of IPv4 tables because the net delay for interconnecting various FPGA components is lighter for small routing tables than large routing tables. We also analyze the memory usages for larger synthesized IPv6 tables generated by V6GEN [44]. The largest table that needs the memory storage of 24,586 Kb affordable by Virtex-6 XC6VSX315T is the one consisting of 290,503 distinct entries.

6 Conclusion

In this paper, we proposed and implemented a pipeline design called LayeredTrees for IP address lookups. LayeredTrees consists of multi-layered multiway balanced prefix trees. In order to store the entire routing table in the on-chip memory, LayeredTrees is optimized by the lite prefix representation, the segmentation tables, the concept of aggregate array along with the base address, and four techniques. The proposed Leveled Search Engine (LSE) is a 5-stage pipeline and the parallel LSE uses 17-stage pipelines. The performance experiments on the chip XC6VSX315T of Virtex-6 FPGA family show that achieved throughput by parallel LSE is superior compared to the existing designs. Also, the proposed LayeredTrees can support the IPv6 routing tables of size as large as 290,503 distinct entries on the same FPGA device.

References

[1] F. Baboescu, D.M. Tullsen, G. Rosu, and S. Singh, "A Tree Based Router Search Engine Architecture with Single Port Memories," *Proc. IEEE Int'l Symp. Computer Architecture (ISCA)*, pp. 123-133, 2005.

[2] F. Baker, Requirements for IP Version 4 Routers, *IETF RFC 1812*, June 1995; www.rfc-editor.org/rfc/rfc1812.txt.

[3] M. Bando, S. Artan, and H.J. Chao, "FlashLook: 100Gbps Hash-Tuned Route Lookup Architecture," *Proc. Int'l Workshop on High Performance Switching Routing (HPSR)*, pp. 1-8, 2009.

[4] M. Bando and H.J. Chao, "FlashTrie: Hash-Based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps," *Proc. 29th IEEE Int'l Conf. Computer Comm. (IEEE INFOCOM)*, pp. 1-9, 2010.

[5] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines," *Proc. 22th IEEE Int'l Conf. Computer Comm.*, pp. 2509-2517, 2003.

[6] M. Behdadfar, H. Saidi, H. Alaei, and B. Samari, "Scalar Prefix Search: A New Route Lookup Algorithm for Next Generation Internet," *Proc. 28th IEEE Int'l Conf. Computer Comm. (IEEE INFOCOM)*, 2009.

[7] *BGP Routing Table Analysis Reports* [Online]. Available: http://bgp.potaroo.net/.

[8] Y.-K. Chang and Y.-C. Lin, "Dynamic Routing Tables Using Simple Balanced Search Trees," *Proc. Int'l Conf. Information Networking (ICOIN)*, pp. 389-398, Jan. 2006.

[9] Y.-K. Chang and Y.-C. Lin, "Dynamic Segment Trees for Ranges and Prefixes," *IEEE Trans. Computers*, vol. 56, no. 6, pp. 769-784, June 2007.

[10] Y.-K. Chang, "Fast Binary and Multiway Prefix Searches for Packet Forwarding," *Computer Network*, vol. 51, no. 3, pp. 588-605, 2007.

[11] Y.-K. Chang, Y.-C. Lin, and C.-C. Su, "Dynamic Multiway Segment Tree for IP Lookups and the Fast Pipelined Search Engine," *IEEE Trans. Computers*, vol. 59, no. 4, pp. 492-506, Apr. 2010.

[12] H.J. Chao, "Next Generation Routers," *Proc. IEEE*, vol. 90, no. 9, pp. 1518-1558, Sep. 2002.

[13] Z. Chicha, L. Milinkovic, and A. Smiljanic, "FPGA Implementation of Lookup Algorithms," *Proc. Int'l Workshop High Performance Switching and Routing (HPSR)*, pp. 270-275, 2011.

[14] Z. Cica and A. Smiljanic, "Frugal IP Lookup Based on a Parallel Search," *Proc. 15th Int'l Workshop on High Performance Switching and Routing (HPSR)*, pp. 1-6, 2009.

[15] S. Demetriades, M. Hanna, S. Cho, and R. Melhem, "An Efficient Hardware-Based Multi-Hash Scheme for High Speed IP Lookup," *Proc. 16th Ann. IEEE Symp. High Performance Interconnects (HOTIC)*, pp. 103-110, 2008.

[16] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *ACM SIGCOMM Computer Comm. Review*, vol. 34, no. 2, pp. 97-122, 2004.

[17] P. Gupta, S. Lin, and N. McKeown, “Routing Lookups in Hardware at Memory Access Speeds,” *Proc. 17th IEEE Int’l Conf. Computer Comm. (IEEE INFOCOM)*,vol. 3, pp. 1240-1247, 1998.

[18] J. Hasan and T.N. Vijaykumar, “Dynamic Pipelining: Making IP Lookup Truly Scalable,” *Proc. ACM SIGCOMM*, pp. 205-216, 2005.

[19] S.-Y. Hsieh, Y.-L. Huang, and Y.-C. Yang, “Multiprefix Trie: A New Data Structure for Designing Dynamic Router-Tables,” *IEEE Trans. Computers*, vol. 60, no. 5, pp. 693-706, June 2011.

[20] W. Jiang and V.K. Prasanna, “A Memory-Balanced Linear Pipeline Architecture for Trie-Based IP Lookup,” *Proc. IEEE Symp. High-Performance Interconnects*, pp. 83-90, 2007.

[21] W. Jiang and V.K. Prasanna, “Multi-Terabit IP Lookup Using Parallel Bidirectional Pipelines,” *Proc. ACM Int’l Conf. Computing Frontiers (CF)*, pp. 241-250, 2008.

[22] W. Jiang, Q. Wang, and V.K. Prasanna, “Beyond TCAMs: An SRAM Based Parallel Multi-Pipeline Architecture for Terabit IP Lookup,” *Proc. IEEE INFOCOM*, pp. 2458-2466, Apr. 2008.

[23] S. Kumar, M. Becchi, P. Crowley, and J. Turner, “CAMP: Fast and Efficient IP Lookup Architecture,” *Proc. ACM/IEEE Symp. Architectures for Networking and Comm. Systems (ANCS)*, pp. 51-60, 2006.

[24] H. Le, W. Jiang, and V.K. Prasanna, “A SRAM-Based Architecture for Trie-Based IP Lookup Using FPGA,” *Proc. Int’l Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 33-42, 2008.

[25] H. Le, W. Jiang, and V.K. Prasanna, “Scalable High Throughput SRAM Based Architecture for IP Lookup Using FPGA,” *Proc. IEEE Int’l Conf. Field Programmable Logic and Applications (FPL)*, pp. 137-142, 2008.

[26] H. Le and V.K. Prasanna, “Scalable High Throughput and Power Efficient IP-Lookup on FPGA,” *Proc. 17th IEEE Ann. Int’l Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 167-174, 2009.

[27] H. Le, W. Jiang, and V.K. Prasanna, “Memory-Efficient IPv4/v6 Lookup on FPGAs Using Distance-Bounded Path Compression,” *Proc. 19th IEEE Ann. Int’l Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 242-249, 2011.

[28] H. Le and V.K. Prasanna, “Scalable Tree-Based Architectures for IPv4/v6 Lookup Using Prefix Partitioning,” *IEEE Trans. Computers*, vol. 61, no. 7, pp. 1026-1039, July 2012.

[29] H. Lu and S. Sahni, “Enhanced Interval Tree for Dynamic IP Router-Tables,” *IEEE Trans. Computers*, vol. 53, no. 12, pp. 1615-1628, Dec. 2004.

[30] H. Lu and S. Sahni, “O(logn) Dynamic Router-Tables for Prefixes and Ranges,” *IEEE Trans. Computers*, vol. 53, no. 10, pp. 1217-1230, Oct. 2004.

[31] H. Lu, K. Kim, and S. Sahni, “Prefix and Interval-Partitioned Dynamic IP Router-Tables,” *IEEE Trans. Computers*, vol. 54, no. 5, pp. 545-557, May 2005.

[32] H. Lu and S. Sahni, “A B-Tree Dynamic Router-Table Design,” *IEEE Trans. Computers*, vol. 54, no. 7, pp. 813-824, July 2005.

[33] Y. Luo, L. Bhuyan, and X. Chen, “Shared Memory Multiprocessor Architectures for Software IP Routers,” *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 12, pp. 1240-1249, Dec. 2003.

[34] D. Pao, Z. Lu, and Y.H. Poon, “Bit-Shuffled Trie: IP Lookup with Multi-Level Index Tables,” *Proc. IEEE Int’l Conf. Comm. (ICC)*, 2011.

[35] *RIPE RIS Raw Data* [Online]. Available: http://www.ripe.net/projects/ris/rawdata.htmlaccessed on Feb. 20, 2014.

[36] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, “Survey and Taxonomy of IP Address Lookup Algorithms,” *IEEE Network*, vol. 15, no. 2, pp. 8-23, Mar./Apr. 2001.

[37] S. Sahni and K Kim, “An O(logn) Dynamic Rrouter-Table Design,” *IEEE Trans. Computers*, vol. 53, no. 3, pp. 351-363, Mar. 2004.

[38] P. Shivakumar and N. Jouppi, *Cacti*, http://www.hpl.hp.com/research/cacti/accessed on Feb. 20, 2014.

[39] V. Srinivasan and G. Varghese, “Fast Address Lookups Using Controlled Prefix Expansion,” *Proc. ACM Trans. Computer Systems*, vol. 17, no. 1, pp. 1-40, Feb. 1999.

[40] X. Sun and Y.Q. Zhao, “An On-Chip IP Address Lookup Algorithm,” *IEEE Trans. Computers*, vol. 54, no. 7, pp. 873-885, July 2005.

[41] P. Warkhede, S. Suri, and G. Varghese, “Multiway Range Trees: Scalable IP Lookup with Fast Updates,” *Computers Networks*, vol. 44, no. 3, pp. 289-303, Feb. 2004.

[42] Xilinx, Virtex-6 Family Overview, DS150 (v2.3), http://www.xilinx.com/products/silicon-devices/fpga/virtex-6/index.htm, Mar. 24, 2011.

[43] Y.-H.E. Yang and V.K. Prasanna, “High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA,” *Proc. ACM/SIGDA Int’l Symp. Field Programmable Gate Arrays (FPGA)*, 2010.

[44] K. Zheng and B. Liu, “V6Gene: A Scalable IPv6 Prefix Generator for Route Lookup Algorithm Benchmark,” *Proc. Int’l Conf. Advanced Information Networking and Applications (AINA)*, pp. 6-11, 2006.

Yeim-Kuan Chang received the PhD degree in computer science from Texas A&M University, College Station, in 1995. He is currently a professor in the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan. His research interests include Internet router design, computer architecture, and multiprocessor systems.

Fang-Chen Kuo received the MS degree in computer science and information engineering from National Cheng Kung University, Tainan, Taiwan, in 2006. He is currently working toward the PhD degree in computer science and information engineering from National Cheng Kung University. His current research interests include high-speed networks and high-performance internet router design.

Han-Jhen Kuo received the MS degree in computer science and information engineering from National Cheng Kung University, Tainan, Taiwan, in 2010. Her research interests include high-speed packet processing in hardware. From August 2010, she has served as an engineer at the Taiwan Semiconductor Manufacturing Company (TSMC), Hsinchu, Taiwan and is responsible for developing the manufacturing information technology systems.

Cheng-Chien Su received the MS and PhD degrees in computer science and information engineering from National Cheng Kung University, Tainan, Taiwan, in 2005 and 2011, respectively. His research interests include high-speed packet processing in hardware and deep packet inspection architectures.