Scalable Tree-based Architectures for IPv4/v6 Lookup Using Prefix Partitioning

Hoang Le, *Student Member, IEEE,* and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—Memory efficiency and dynamically updateable data structures for Internet Protocol (IP) lookup have regained much interest in the research community. In this paper, we revisit the classic tree-based approach for solving the longest prefix matching (LPM) problem used in IP lookup. In particular, we target our solutions for a class of large and sparsely-distributed routing tables, such as those potentially arising in the next-generation IPv6 routing protocol. Due to longer prefix lengths and much larger address space, preprocessing such routing tables for tree-based LPM can significantly increase the number of prefixes and/or memory stages required for IP lookup. We propose a prefix partitioning algorithm (DPP) to divide a given routing table into k groups of disjoint prefixes (k is given). The algorithm employs dynamic programming to determine the optimal split lengths between the groups to minimize the total memory requirement. Our algorithm demonstrates a substantial reduction in the memory footprint compared with those of the state-of-the-art in both IPv4 and IPv6 cases. Two proposed linear pipelined architectures, which achieve high throughput and support incremental updates, are also presented. The proposed algorithm and architectures achieve a memory efficiency of 1 byte of memory for each byte of prefix for both IPv4 and IPv6. As a result, our design scales well to support either larger routing tables, longer prefix lengths, or both. The total memory requirement depends solely on the number of prefixes. Implementations on 45 nm ASIC and a state-of-the-art FPGA device (for a routing table consisting of 330K prefixes) show that our algorithm achieves 980 and 410 million lookups per second, respectively. These results are well suited for 100Gbps lookup. The implementations also scale to support larger routing tables and longer prefix length when we go from IPv4 to IPv6. Additionally, the proposed architectures can easily interface with external SRAMs to ease the limitation of on-chip memory of the target devices.

Index Terms—IP Lookup, Longest Prefix Matching, Reconfigurable, Field Programmable Gate Array (FPGA), Pipeline, Partitioning.

1 Introduction

IP packet forwarding, or simply, IP-lookup, is a classic problem. In computer networking, a routing table is a database that is stored in a router or a networked computer. The routing table stores the routes and metrics associated with those routes, such as next hop routing indices, to particular network destinations. The IP-lookup problem is referred to as "*longest prefix matching*" (LPM), which is used by routers in IP networking to select an entry from the given routing table. To determine the outgoing port for a given address, the longest matching prefix among all the prefixes needs to be determined. Routing tables often contain a default route in case matches with all other entries fail.

With the rapid growth of the Internet, IP-lookup becomes the bottle-neck in network traffic management. Therefore, the design of high speed IP routers has been a major area of research. High link rates demand that packet forwarding in IP routers must be performed in hardware [23]. For instance, a 100 Gbps link requires a throughput of over 150 million lookups per second. Such line rates demand fast lookup algorithms with *compact data structures*.

At the core routers, the size of the routing table also increases at the rate of 25-50K prefixes per year [8]. Additionally, IPv6 extends the size of the IP address from 32 bits found in IPv4 to 128 bits. This increased size allows for a broader range of addressing hierarchies and a much larger number of addressable nodes. Although the length of an IPv6 address is 128 bits, according to the Internet Architecture Board (IAB), an IPv6 address consists of two parts: a 64-bit network/sub-network ID followed by a 64-bit host ID. Therefore, only the 64-bit network/sub-network ID is relevant in making the forwarding decisions. The increase in the size of routing table and the extension of the prefix length necessitate high *memory-efficient* lookup algorithms to reduce the size of the storage memory. Moreover, the use of multi-core and network processors demands compact memory footprints that can fit in on-chip caches to ensure high throughput. The emergence of network virtual routers also stretches the constraints on per-virtual-router storage memory to reduce the total memory requirement. Note that for virtual routers, shared data structures such as [12] and [29], can be used to reduce memory usage. However, our focus in this paper is on the isolated case when each router maintains its own routing table.

Our analysis of the synthetic IPv6 tables, which were generated using [33], suggests that current solutions for IP lookups do not actually scale very well to support IPv6. The most popular IP lookup algorithms are trie-based. However, their memory efficiency decreases linearly as the tree depth increases from 32 bits in IPv4 to 128 bits in IPv6, making the trie-based solutions less

• *H. Le and V. K. Prasanna are with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90089. E-mail: {hoangle, prasanna}@usc.edu*

Supported by the U.S. National Science Foundation under grant No. CCF-1018801. Equipment grant from Xilinx is gratefully acknowledged.

attractive. Path compression techniques ([20], [27]) can greatly help at the cost of increase in the computational complexity at the nodes. On the other hand, traditional approaches to partition the prefix table into set(s) of disjoint prefixes (required for tree-based solutions), such as leaf-pushing [32] and prefix expansion [32], also suffer from the high prefix expansion issues in IPv6.

Most hardware-based solutions in network routers fall into two main categories: TCAM-based and dynamic/static random access memory (DRAM/SRAM)-based solutions. Although TCAM-based engines can retrieve results in just one clock cycle, their throughput is limited by the relatively *low speed* of TCAMs. They are expensive, *power-hungry*, and offer little adaptability to new addressing and routing protocols [2]. SRAM-based solutions, in contrast, require multiple cycles to process a packet. Therefore, pipelining techniques are commonly used to improve the throughput. These SRAM-based approaches, however, result in an inefficient memory utilization. This inefficiency limits the size of the supported routing tables. In addition, it is not feasible to use external SRAM in these architectures, due to the limitation on the number of I/O pins. This constraint restricts the number of external stages, while the amount of on-chip memory confines the size of memory for each pipeline stage. Due to these two constraints, state-of-the-art SRAM-based solutions do not scale to support larger routing tables. This scalability has been a dominant issue in the existing implementations. Furthermore, pipelined architectures increase the total number of memory accesses per clock cycle; and thus increase the dynamic power consumption. The power dissipation in the memory dominates that in the logic [16], [21], [15]. Hence, reducing memory power dissipation contributes to a large reduction in the total power consumption.

Our focus in this paper is on achieving significant reduction in memory requirements for the longest prefix-match operation needed for IPv4/v6 lookup. Fast and efficient IP lookup has been well-studied in the research community, and numerous algorithms have been proposed (more details in Section 2.2). However, it is still possible to achieve a substantial reduction in memory usage. This paper makes the following contributions:

- An algorithm that partitions a routing table into groups of prefixes to *minimize* the total memory consumption (Section 4).
- A data structure based on a complete binary search tree that achieves *high* memory efficiency of 1 byte of memory per byte of prefix for both IPv4 and IPv6 routing tables (Section 5.1).
- A data structure based on 2-3-tree that supports *single-cycle* incremental update (Section 5.2).
- Dual linear pipelined architectures that use dual-ported memory to achieve high throughput (Section 6).
- A design that supports the largest routing table consisting of over $330K$ IPv4/v6 prefixes and sustains a high throughput of 980 and 410 million lookups per second on 45 nm ASIC and a state-of-the-art FPGA device, respectively. External SRAM can also easily be utilized to ease the limitation of the amount of on-chip memory and the number of I/O pins (Section 9).

TABLE 1: A sample routing table (Maximum prefix length = 8)

	Prefix	Next Hop		Prefix	Next Hop
P_0	*	0	P_5	11111*	5
P_1	000*	1	P_6	010*	6
P_2	01000*	2	P_7	1*	7
P_3	01011*	3	P_8	0*	8
P_4	11010*	4			

The rest of the paper is organized as follows. Section 2 introduces the background and related work. Section 3 presents the problem definition. Section 4 details the prefix partitioning algorithm. Section 5 covers the BST-based and 2-3-tree-based IP lookup algorithms. Section 6 and 7 describe the proposed architectures and their implementation. Section 8 provides the proposed power optimization techniques. Section 9 presents the performance evaluations. Section 10 concludes the paper.

2 Background and Related Work

2.1 Background

A sample routing table with the maximum prefix length of 8 is illustrated in Table 1. This sample table will be used throughout the paper. Each prefix is associated with a next hope index value, which indicates the corresponding outgoing port. Note that the next hop values need not be in any particular order. In this routing table, binary prefix P_3 ($01011*$) matches all destination addresses that begin with 01011. Similarly, prefix P_4 matches all destination addresses that begin with 11010. The 8-bit destination address **IP**$= 01000000$ is matched by the prefixes P_0, P_2, P_6, and P_8. Since $|P_0| = 0$, $|P_2| = 5$, $|P_6| = 3$, $|P_8| = 1$, P_2 is the longest prefix that matches **IP** ($|P|$ is defined as the length of prefix P, in bits). In longest-prefix routing, the next hop index for a packet is given by the table entry corresponding to the longest prefix that matches its destination IP address. Thus, in the above example, the next hop of 2 is returned.

2.2 Related Work

In general, algorithms for IP-lookup can be classified into the following categories: trie-based, scalar-tree-based, range-tree-based, and hash-based approaches. Each has its own advantages and disadvantages.

In *trie-based approaches* ([4], [17], [30], [14], [3], [31]), IP-lookup is performed by simply traversing the trie according to the bits in the IP address. These designs have a *compact data structure* but a *large* number of trie nodes; hence moderate memory efficiency. Furthermore, the latency of these trie-based solutions is proportional

to the prefix length, making them less attractive for IPv6 protocol, unless multi-bit trie is used, as in [30]. This architecture employs a method called *shape graph* to achieve good memory efficiency and high throughput. However, it requires more complex preprocessing. The optimization also makes it difficult to update. Additionally, the use of hash table for next-hop lookup demands a highly efficient hash function to avoid collisions, which can dramatically reduce the overall performance. For devices with the limited amount of on-chip memory, it is also difficult to utilize external memory due to large number of pipeline stages and limited number of I/O pins.

In *range-tree-based approaches* ([19], [34]), each prefix is treated as a *range*. In [19], prefixes are converted to non-intersecting ranges, and a B-tree data structure is used to search these ranges. The number of ranges is twice the number of prefixes in the worst case. The main advantage of this approach is that each prefix is stored in $O(1)$ B-tree nodes per B-tree level. Thus, updates can be performed in $O(\log_m n)$, where m is the order of the B-tree and n is the number of prefixes in the routing table. In [34], IP lookup is performed using a multi-way range tree (a type of B-tree) algorithm, which achieves worst-case search and update time of $O(\log n)$. The proposed approach achieves the optimal lookup time of binary search, and can be updated in logarithmic time. Another advantage of these tree-based approaches is that the lookup latency does not depend on the prefix length, but is proportional to the logarithm of the number of prefixes; therefore, they are highly suitable for IPv6. Note that a B-tree of order greater than 3 is very difficult to be efficiently implemented on hardware due to (1) the poor uniformity in the data structure and (2) the wide-memory access required for each fat node.

The *scalar-tree-based approaches* were also proposed ([18], [5]). In [18], each prefix is treated as a search key and the entire routing table is stored into multiple binary search trees. This approach achieved low memory footprint at the cost of high memory bandwidth. In [5], the authors presented the "Scalar Prefix Search", which interprets each prefix as a number without any encoding. Using keys and match vectors, it can store several prefixes in one key. This approach reduced the number of prefix keys while doubling the memory bandwidth. The memory requirement also increases for routing tables with high percentage of distinct prefixes.

In *hash-based approaches* ([30], [28], [26], [11]), hashing functions or bloom filters are used. The architecture in [11] takes advantage of the benefit of both a traditional hashing scheme and reconfigurable hardware. However, these architectures only result in moderate memory efficiency, and can only be updated using partial reconfiguration when adding or removing prefixes. It is also unclear how to scale these designs to support larger routing tables and/or longer prefix lengths. Additionally, they suffer from non-deterministic performance due to hash collisions.

3 Problem Definition

The problem of partitioning and searching a routing table is defined as follows. Given a routing table R consisting of N prefixes, and a scalar number k, find (1) an algorithm that can efficiently partition this table into k groups of disjoint prefixes, namely $\{G_1, G_2, ...G_k\}$, to minimize the total memory requirement and (2) a parallel search data structure that can support high throughput and quick update.

4 Prefix Partitioning Algorithm

4.1 Definitions

The following definitions and notations are used throughout this paper:

Definition 1: Two distinct prefixes are said to be ***overlapped*** if and only if one is a proper prefix of the other. Otherwise, they are said to be ***disjoint***. For instance, in Table 1, P_1 and P_8 are overlapped, P_1 and P_2 are disjoint.

Definition 2: A set of prefixes is considered ***disjoint*** if and only if any 2 prefixes in the set do not overlap with each other.

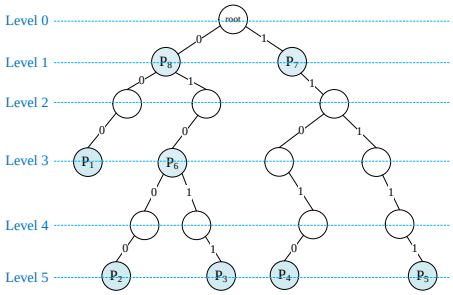
Definition 3: The ***memory efficiency*** is defined as the amount of memory (in bytes) required to store one byte of prefix.

Definition 4: The memory footprint is defined as the size of the memory required to store the entire routing table. The terms ***memory requirement***, ***memory footprint***, and ***storage memory*** are used interchangeably in this paper.

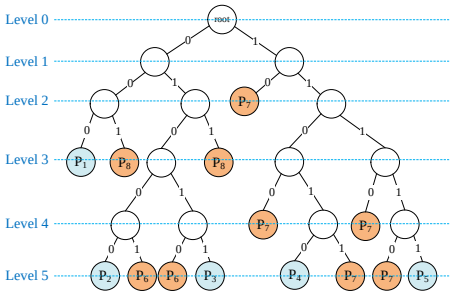
4.2 Prefix Paritioning Algorithm

Tree search algorithm is a good choice for IPv6 forwarding engine as the lookup latency does not depend on the length of the prefix, but on the number of prefixes in the routing table. In case of the tree search, the latency is proportional to $\log$ of the number of prefixes. Note that the trie-based approaches can also reduce the latency by using multi-bit trie. However, this reduction comes at the cost of memory explosion; thus, resulting in a very *poor* memory efficiency. Furthermore, as mentioned before, the number of nodes in the trie drastically expands as the prefix length increases from 32 bits to 128 bits. Path compression techniques ([20], [27]) work well to contain the number of trie nodes. Yet, they increase the computational complexity at the nodes and reduce the look-up performance.

In this paper, the focus is on the classic *scalar tree-based* approach for IP lookup. In order to use tree search algorithms, the given set of prefixes needs to be processed to eliminate the overlap between prefixes. This elimination process results in a set (or sets) of *disjoint* prefixes. There are many approaches proposed to eliminate overlap, such as leaf-pushing [32], prefix expansion [32], and prefix merging [5]. Among these approaches, leaf-pushing is the most popular one due to its simplicity. However, as previously stated, these



(a) Sample trie



(b) Leaf-pushed trie

Fig. 1: A sample trie and its leaf-pushed trie

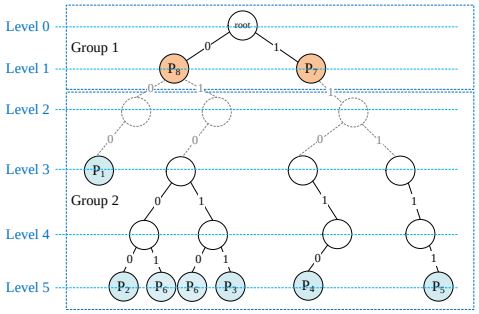


Fig. 2: 2-level leaf-pushed trie

TABLE 2: Prefixes in each group

Group	Prefix	Padded prefix	Next hop
G_1	0*	00000000	8
	1*	10000000	7
G_2	000*	00000000	1
	01000*	01000000	2
	01001*	01001000	6
	01010*	01010000	6
	01011*	01011000	3
	11010*	11010000	4
	11111*	11111000	7

approaches either increase the number of prefixes, the number of sets of disjoint prefixes, or both. What we are interested in is an algorithm that can minimize the total amount of memory required, for a given fixed number of sets.

Leaf-pushing by itself is a simple idea. In this approach, the initial step is to build a trie from the given routing table. Leaf pushing first grows the trie to a full tree (i.e. all the non-leaf nodes have two child nodes), and then pushes all the prefixes to the leaf nodes. Fig. 1(a) and 1(b) illustrate the trie of the sample routing table shown in Table 1 and its leaf-pushed trie, respectively. All the leaf nodes are then collected to build a *search tree*. Since the set of leaf nodes are disjoint, any tree search algorithm can be employed to perform the lookup. While leaf pushing eliminates overlap between prefixes, it has the negative effect of expanding the size of the prefix table. For all publicly available IPv4 routing tables, the prefix tables expand about 1.6 times after leaf pushing [30].

Instead of performing leaf-pushing for the entire trie, *multi-level leaf-pushing* can be utilized. The following notations are used in the context:

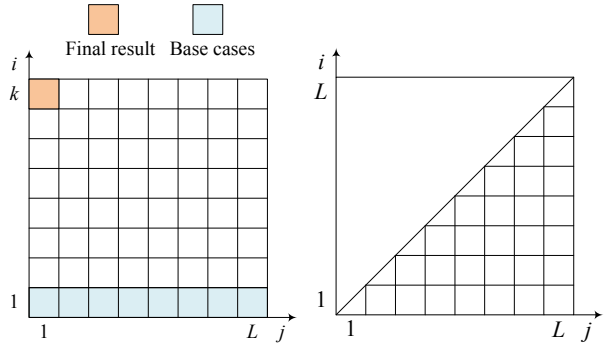
1) k: the number of target leaf-pushed prefix levels
2) $G_1, G_2, ...G_k$: k groups of disjoint prefixes
3) L: the maximum prefix length (32 in IPv4 and 128 in IPv6)

The *multi-level leaf-pushing* includes 3 steps. First, the target prefix levels, $L_1, L_2, ...L_{k-1}$, are determined. Secondly, prefixes with length in $(L_i, L_{i+1}]$ are leaf-pushed to length L_{i+1}. Thirdly, all the leaf-pushed prefixes in each length range are collected into their corresponding groups $((0, L_1] \rightarrow G_1, (L_1, L_2] \rightarrow G_2, ...(L_{k-1}, L] \rightarrow G_k)$. Note that reduction in the total number of prefixes does not necessarily lead to reduction in the size of storage memory. When we build a search data structure, the sizes of the entry are identical within a group, but different between groups. The ultimate goal is to pick $L_1, L_2, ...L_{k-1}$ such that the total memory consumption is minimized. Therefore, the entry's size should be taken into consideration. Also, when $k = L$, the problem becomes trivial as $L_i = i$. In this case, group G_i contains all the prefixes of length i, and the total number of prefixes in all groups is exactly equal to the number of prefixes in the given routing table.

The sample trie in Fig. 1(a) is used as an example. All the prefix nodes are colored. Fig. 1(b) shows the regular leaf-pushed trie, whereas Fig. 2 depicts the 2-level leaf-pushed trie. In the first case, *one* group of 14 disjoint prefixes is generated. In the second case, the trie is leaf-pushed to levels $1, 5$. Consequently, *two* groups of disjoint prefixes (G_1, G_2) are created, with only a total of 9 prefixes. The prefix distribution in each group is shown in Table 2. The drawback of this approach is the increased number of groups (search data structures). However, this is not a concern in hardware implementation.

The total number of possibilities of choosing the target levels is $L!/((L-k)! \times k!)$. We chose the optimal target levels using *dynamic programming* [10]. Note that recursion can also be used to solve the problem. However, it can lead to repeated subproblems. Dynamic programming helps avoid this repetition by explicitly enumerating the distinct subproblems and solving them in the correct order.

We select $k-1$ target levels, $L_1, L_2, ...L_{k-1}$, to partition the given routing table as follows. First, L_1 is assigned with all the possibilities. Since each group must contain



(a) Cost table $C(i,j), i \leq j$ (b) Expansion table $E(i,j), i \leq j \leq L$

Fig. 3: Steps in the proposed partitioning algorithm

TABLE 3: Notations used in dynamic programming equations

Notation	Meaning
$PLP(i,j),\ i \leq j$	*Partial-leaf-pushing* function, which leaf-pushes all the prefixes with lengths in $[i,j]$ up to level j. In other words, if a trie is built for these prefixes, then the PLP function will simply leaf-push the trie and return the number of leaf nodes
S_{entry}	Size of an entry in each group. Each entry consists of a prefix (L bits), a prefix length ($\lceil \log L \rceil$ bits), and a next hop information (h bits). If the goal is to minimize the total number of prefixes, then S_{entry} is set to 1
$E(i,j),\ i \leq j$	Expansion cost from level i to level j
$C(k,pos)$	Cost function to partition the input routing table into k groups, starting from level pos, and perform the leaf-pushing on each group of prefixes. This is the total memory consumption if $S_{entry} \neq 1$, or the total number of leaf-pushed prefixes otherwise

at least one level, it is obvious that $L_1 \in [1, L-k+1]$, where L denotes the maximum prefix length. For each value of L_1, the cost of selecting the remaining $k-2$ target levels C and the cost to expand (or leaf-push) the first prefix partition E are computed. The optimal cost of partitioning the routing table is obtained by choosing the value of L_1 such that $C+E$ is minimized. Therefore, the problem of selecting $k-1$ target levels is divided into $L-k+1$ subproblems of selecting $k-2$ target levels, $L_2, ...L_{k-1}$. The optimality of the solution of the overall problem depends on that of the solution of the subproblems, and in turn, on their subproblems, recursively. Let $C(m, len)$ denote the optimal cost to leaf-push all the prefixes with length len or longer of a given routing table using m target levels. The partitioning algorithm can be formulated by the recursion given in (1). The iterations of this algorithm are described in Fig. 3. The notations used in this equation are defined in Table 3.

The base cases (first row in Fig. 3(a)) are computed using (3). In each iteration, the expansion cost E is computed using (2). These expansion costs can be pre-computed and stored in an expansion table (as shown in Fig. 3(b)). Once the memory consumption is calculated for each value of l, the final levels can be obtained. The pseudo code of the algorithm is shown in Algorithm 1.

$$C(m, len) = \min_{i=len+1}^{L-m+1} \{C(m-1, i) + E(len, i-1)\} \qquad (1)$$
$$1 \leq len \leq L$$
$$E(i,j) = PLP(i,j) \times S_{entry} \qquad i \leq j \qquad (2)$$
$$C(1,i) = E(i,L) \qquad for\ all\ i,\ 1 \leq i \leq L \qquad (3)$$

Algorithm 1 DPP(k, len, L)

Input: A routing table, expansion table E, and k
Output: Optimal target levels and total memory consumption
1: **if** $(k == 1)$ **then**
2: return $E(len, L)$
3: **else**
4: return $\min_{i=len+1}^{L-k+1}\{C(k-1,i) + E(len, i-1)\}$
5: **end if**

4.3 Complexity of the partitioning algorithm

There are $k \times L$ elements in the cost table (Fig. 3(a)). Each element is computed in $O(L)$ time by taking the minimum over at most L quantities. Therefore, the overall complexity of the partitioning algorithm is $O(k \times L^2)$.

5 IP Lookup Algorithms

With the given routing table partitioned into k groups of *disjoint* prefixes, any tree search algorithms can be used to perform the lookup in each group. The final result (longest match) is selected from matches coming out of these k groups (at most one match per group). Two algorithms are presented in this section, one is focusing on *high memory efficiency* (complete binary search tree) and the other is on *incremental update* (2-3-tree).

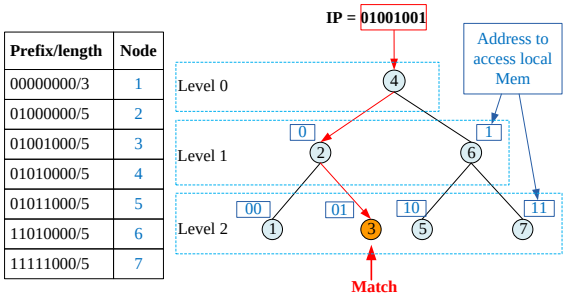
5.1 BST-based IP Lookup

In this section, a memory efficient data structure based on a binary search tree (BST) [6] is presented. BST is a special binary tree data structure with the following properties:

1) Each node has a value
2) The left subtree of a node contains only values less than the node's value
3) The right subtree of a node contains only values greater than the node's value

The binary search algorithm is a technique to find a specific element in a sorted list. In a balanced binary search tree, an element, if it exists, can be found in at most $(1 + \lfloor \log_2 N \rfloor)$ operations, where N is the total number of nodes in the tree.

A binary search tree T_i is built for each group G_i of disjoint prefixes. Note that in each group, shorter prefixes are *zero-padded* up to the length of the longest prefix in that group. Each node of the binary tree contains 3 fields: (1) a padded prefix, (2) the prefix length l, and (3) the corresponding next hop index. Given such a binary



Prefix/length	Node
00000000/3	1
01000000/5	2
01001000/5	3
01010000/5	4
01011000/5	5
11010000/5	6
11111000/5	7

Fig. 4: A sample group of disjoint prefixes and its corresponding complete binary search tree

search tree, IP lookup is performed by traversing left or right, depending on the comparison result at each node. If the entering IP address is less than a node's prefix, it is forwarded to the left branch of the node, and to the right branch otherwise. The matching operation is also performed at each node, by comparing l most significant bits of the incoming prefix and those bits of the node's padded prefix.

The prefixes and the corresponding BST of group G_2 (Fig. 2) are illustrated in Fig. 4. The prefixes in the table are padded with zero(s). For simplicity, 8-bit IP addresses are considered throughout the paper. The IP address enters the tree from its root. At each node, the IP address and node's prefix are compared to determine the matching status and the traversal direction. For example, assume that a packet with destination address of 01001001 arrives. At the root, the prefix is compared with 01010000, which is smaller. Thus, the packet traverses to the left. The comparison with the prefix in node #2 yields a greater outcome; hence, the packet traverses to the right. At node #3, the packet header matches the node's prefix, which is the final result. The similar lookup is performed in each tree T_i. The longest of all the matches from these k trees is returned as the final result.

Algorithm 2 COMPLETEBST(SORTED_ARRAY)

Input: Array $A[\,]$ consists of N prefixes sorted in ascending order
Output: Complete BST
1: $n = \lceil \log_2(N+1) \rceil$
2: $\Delta = N - (2^{n-1} - 1)$
3: **if** $(\Delta \leq 2^{n-1}/2)$ **then**
4: $\quad x = 2^{n-2} - 1 + \Delta$
5: **else**
6: $\quad x = 2^{n-1} - 1$
7: **end if**
8: Pick element $A[x]$ as root
9: Left-branch = COMPLETEBST(left-of-x sub-array)
10: Right-branch = COMPLETEBST(right-of-x sub-array)

We use a *complete* BST for efficient memory utilization. A complete BST is a binary tree in which all the levels are fully occupied except possibly the last level [9]. If the last level is not full, then all the nodes are as far left as possible. Given a set of disjoint prefixes sorted in ascending order, the complete BST can easily be built by picking the correct prefix (pivot) as the root, and recursively building the left and right subtrees. Two

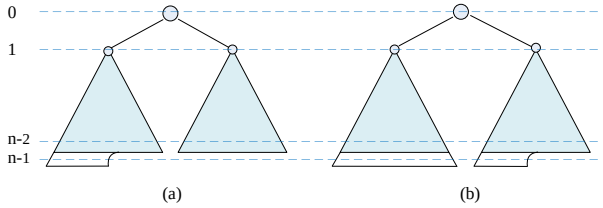


Fig. 5: Two cases of complete BST

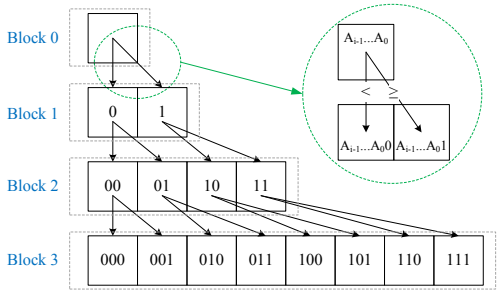


Fig. 6: Pointer elimination in a complete BST

cases of complete BST, in which the last level is not complete, are illustrated in Fig. 5.

Let N_i be the number of prefixes in group G_i, n_i be the number of levels of the BST of G_i, and Δ be the number of prefixes in the last level. The total number of nodes in all stages, excluding the last stage, is $2^{n_i-1} - 1$. Therefore, the number of nodes in the last stage is $\Delta = N_i - (2^{n_i-1} - 1)$. There are 2^{n_i-1} nodes in the last stage if it is full. If $\Delta \leq 2^{n_i-1}/2$, we have a complete BST, as in Fig. 5 (a), or (b) otherwise. Let x be the index of the root of the BST. Since the left subtree of the root contains only values less than the root's value, x can be calculated as $x = 2^{n_i-2} - 1 + \Delta$ for case (a), or $x = 2^{n_i-1} - 1$ for case (b). The complete BST can be built recursively, as described in Algorithm 2.

Pointer Elimination: In a binary search tree, each node contains a data and two child pointers. The memory efficiency can be improved if these pointers are eliminated. The complete BST can be mapped as follows. Nodes in each level are stored in a contiguous memory block. Let x denote the index (in base-2) of node A. A has 2 child nodes with the index of $2x$ (left child) and $2x+1$ (right child). In base-2 number system, $2x = \{x, 0\}$ and $2x+1 = \{x, 1\}$, where $\{\}$ represents the concatenation. Using this memory mapping, there is no need to store child pointers at each node. Instead, these pointers are calculated explicitly on-the-fly by simply concatenating the address of the current node and the comparison result bit. For instance, consider a node with index $A_{i-1}...A_0$ in level i. If the input data is less than the node's data, then the next child pointer is $A_{i-1}...A_0 0$, or $A_{i-1}...A_0 1$ otherwise. The memory allocation and address calculation are described in Fig. 6.

5.2 2-3 Tree-base IP Lookup

The main drawback of the complete-BST-based data structure is the difficulty in supporting updates in *hard-*

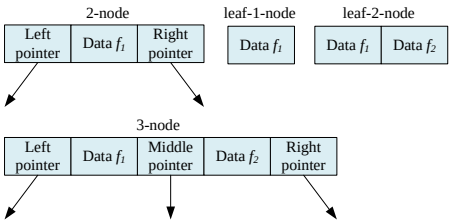


Fig. 7: Different types of nodes of a 2-3 tree

ware (more details in Section 6.4). Therefore, in this section, a memory efficient data structure that can support incremental update is presented. It is based on a 2-3 tree [1], which is a type of B-tree with order of 3. A 2-3 tree is a balanced search tree that has the following properties:

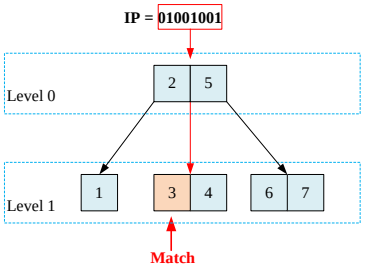
1) There are three different types of nodes: a leaf node, a 2-node and a 3-node (Fig. 7).
2) A leaf node contains one or two data fields.
3) A 2-node is the same as a binary search tree node, which has one data field and references to two children.
4) A 3-node contains two data fields, ordered so that the first is less than the second, and references to three children. One child contains values less than the first data field, another child contains values between the two data fields, and the last child contains values greater than the second data field.
5) All of the leaves are at the lowest level.

A 2-3 tree with N nodes never has a height greater than $\log_2(N+1)$. Traversing in 2-3 tree can be performed in a similar way as in a binary search tree: compare and move down the correct link until a leaf node is reached. The search algorithm in 2-3 tree runs in $O(\log N)$ time, where N is the total number of elements in the tree. Note that the pointers in a 2-3 tree should not be eliminated as they cannot be calculated on-the-fly, in contrast to the complete BST in the previous subsection.

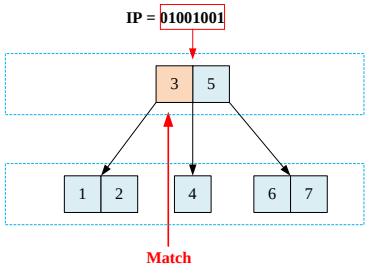
A 2-3 tree is built for each set of disjoint prefixes. Unlike the complete BST, the set of prefixes needs not to be sorted prior to building the 2-3 tree. These prefixes can be added in the tree in any order. Steps to build a 2-3 tree are described in Algorithm 3. Given such a 2-3 tree, IP lookup is performed by traversing left or right, depending on the comparison result at each node. At a 2-node, if the entering IP address is smaller or equal to a node's prefix, it is forwarded to the left branch of the node, and to the right branch otherwise. At a 3-node, if the entering IP address is smaller or equal to the node's left prefix, it is forwarded to the left branch of the node. If the IP address is larger than the node's right prefix, it is forwarded to the right branch of the node, and to the middle branch otherwise.

The prefixes and the corresponding 2-3-tree of group G_2 (Fig. 2) are illustrated in Fig. 8(a). Note that there may possibly be more than one 2-3-tree for a given data set, depending on the order of items to be inserted. For example, Fig. 8(a) and 8(b) represent the same set of prefixes. Each node of the binary tree contains 1 or 2 *prefixes* and their corresponding *next hop indices*, for a 2-node or 3-node, respectively. The IP address enters the tree from its root. At each node, the IP address and node's prefix

Prefix/length	Node
00000000/3	1
01000000/5	2
01001000/5	3
01010000/5	4
01011000/5	5
11010000/5	6
11111000/5	7



(a) Sample 2-3-tree 1



(b) Sample 2-3-tree 2

Fig. 8: A sample group of disjoint prefixes and two valid corresponding 2-3-trees

Algorithm 3 TWOTHREETREE(*root*, PREFIX p)

Input: The *root* of the existing 2-3-tree, new prefix p
Output: Prefix p is inserted into the 2-3-tree

while (Leaf node is not reached) **do**
  Compare p with node's data and move down the correct link.
end while
if (Leaf node has only one data item) **then**
  Convert leaf node to a 3-node
  Insert p into the 3-node
else if (Leaf node is full and the parent node has only one data item) **then**
  Compare the three values: the two leaf node data items and p
  Select the middle value and insert it in the parent node
  if (leaf node is the left child) **then**
    Shift the old data in the parent node to the right
    Insert the middle value in the parent node before the old data
    Shift the pointers to the children in the parent node
  else
    Insert the middle value to the right of the old value in the parent node
    The two left-over values from the comparison become two leaf nodes
  end if
else
  Promote the middle value of the leaf node to its parent node
  while (parent node is full and current node is not *root*) **do**
    Promote the middle value of the current node
    Update current node
  end while
  if (current node is *root*) **then**
    A new root node is created from the middle value to its parent node
    Update the left and right subtrees of the new root
    Update *root* ← new root
  end if
end if

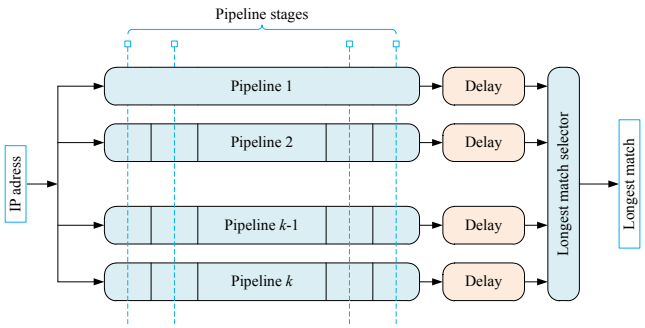


Fig. 9: The overall IP lookup architecture

are compared to determine the matching status and the traversal direction. For instance, assume that a packet with a destination address of $IP = 01001001$ arrives. At the root of the tree in Fig. 8(a), IP is compared with node values 010000000 and 01011000, yielding no match and a "middle" result. Thus, the packet traverses to the middle branch. The comparison with the prefix in this node results in a match, which is the matched result of the tree.

6 Architecture

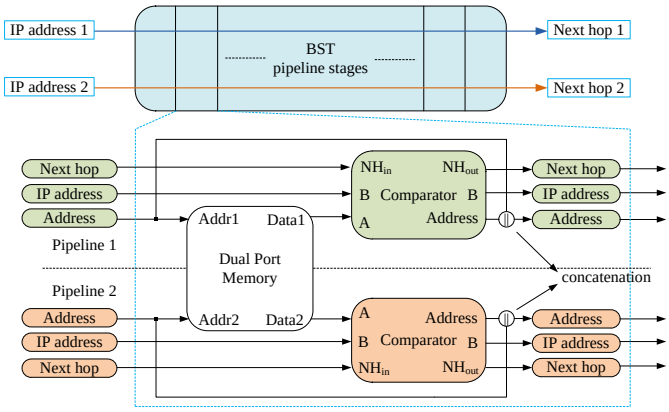
6.1 Overall architecture

Pipelining is used to produce one lookup operation per clock cycle to increase the throughput. The number of pipeline stages is determined by the height of the search tree. Each level of the tree is mapped onto a pipeline stage, which has its own memory (or table).

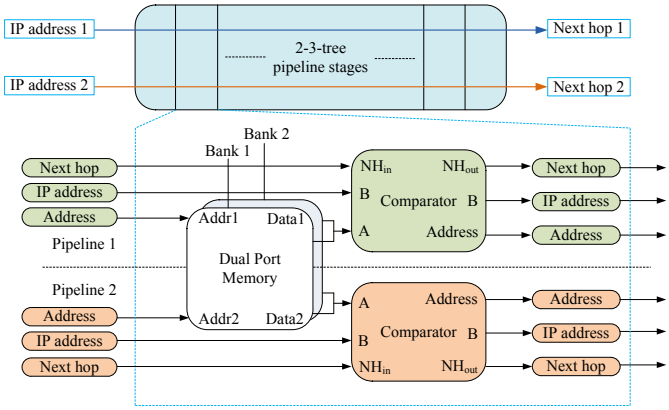
The overall architecture of the proposed tree-based approach is shown in Fig. 9. There are k pipelines, one for each group of prefixes. The number of stages in each pipeline depends on the number of prefixes in each group, and can be easily determined for any given routing table. When a packet arrives at the router, its destination IP address is extracted and routed to all branches. The searches are performed in parallel in all the pipelines. The results are fed through a longest match selector to pick the next hop index of the longest matched prefix. The variation in the number of stages in these pipelines results in latency mismatch. Hence, a delay block is appended to each shorter pipeline to match with the latency of the longest pipeline.

6.2 BST-based Architecture

The block diagrams of the basic pipeline and a single stage are shown in Fig. 10(a). In the proposed architecture, dual-ported memory is utilized. The architecture is configured as dual linear pipelines to double the lookup rate. At each stage, the memory has dual Read/Write ports so that two packets can be input every clock cycle. The content of each entry in the memory includes the prefix and its next hop routing index. In each pipeline stage, there are 3 data fields forwarded from the previous stage: (1) *IP address*, (2) *memory access address*, and (3)



(a) BST



(b) 2-3-tree

Fig. 10: A single stage of the pipeline

next hop index. The forwarded memory address is used to retrieve the node prefix, which is compared with the IP address to determine the matching status. In case of a match, the next hop index of the new match replaces the old result. The comparison result (1 if the IP address is greater than node prefix, 0 otherwise) is appended to the current memory address and forwarded to the next stage. If a match has been found, search in the subsequent stages is unnecessary as all the prefixes within a set are disjoint. Hence, memory access in those subsequent stages can be turned off and the previous result can simply be forwarded to save power consumption.

6.3 2-3-Tree-based Architecture

The block diagram of the single stage of a 2-3-tree pipeline is shown in Fig. 10(b). Similarly to the BST architecture, the 2-3-tree architecture is configured as dual-linear pipelines to take advantage of the dual-ported feature of the memory. At each stage, two IP addresses can be input every clock cycle to double the throughput. In each pipeline stage, there are 3 data fields forwarded from the previous stage: (1) *IP address*, (2) *next hop*, and (3) *memory address*. The memory address is used to retrieve the node stored on the local memory. The node value is compared with the input IP address to

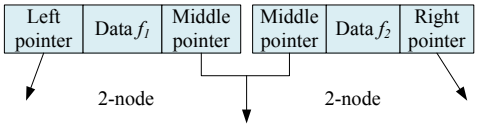


Fig. 11: The modified 3-node

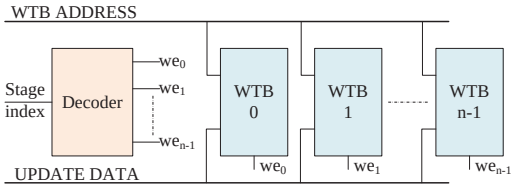


Fig. 12: Block diagram of the architecture to update the write bubble tables (WTBs)

determine the match status and the direction of traversal. The *memory address* is updated in each stage using the correct child address, depending on the direction of traversal. However, the *next hop information* is only updated if a match is found at that stage. In this case, search in subsequent stages is unnecessary as all the prefixes with a set are disjoint. Hence, those subsequent stages can be turned off to save power consumption.

Memory Management in 2-3-tree: The major difficulty in efficiently realizing a 2-3 tree on hardware is the difference in the size of the 2-node and 3-node. The space allocated for a 2-node cannot be reused later by a 3-node. The available memory management is also more complicated. To overcome this problem, a 3-node is divided into two 2-nodes (Fig. 11). The left pointer of the first node points to the left child. The right pointer of the first node and the left pointer of the second node both point to the middle child. Finally, the right pointer of the second node points to the right child. Although a pointer is redundant, the benefit is justifiable. It creates *uniformity* in the size of the nodes, and simplifies the memory management. Additionally, this node-splitting scheme allows us to precisely estimate the amount of required memory, as each prefix is stored in exactly one *effective* 2-node.

Two memory banks are used to support the node-splitting scheme. A 2-node can be placed in any bank, while a 3-node spans over 2 banks. Note that when a 3-node is split into two 2-nodes, each of them must be placed in the same memory location in each bank. This placement allows us to use a single address to access both nodes. Each pipeline stage can be configured to have parallel access to both memory banks in one clock cycle, or have sequential access to these banks in 2 consecutive cycles. In the second case, the latency of the pipeline is doubled, and more logic resources are used.

6.4 Update in BST

Routing table updates include three operations: (1) existing prefix modification, (2) prefix deletion, and (3) new prefix insertion. The first update requires changing the next hop indices of the existing prefixes in the routing table, while the others require inserting a prefix into, or deleting a prefix from a given routing table.

We define two types of updates: *in-place* update and *new-route* update. All updates related to existing prefixes, such as: change flow information, bring up or down a prefix, belong to the first type. The in-place updates can be performed by inserting a write bubble, as introduced in [4]. These in-place updates can easily be done by inserting write bubbles into the lookup traffic, as shown in Fig. 13. There is one dual-ported write bubble table (WBT) in each stage. Each table consists of at least 2 entries. Each entry composes of: (1) the memory address to be updated in the next stage, (2) the new content for that memory location, and (3) a write-enable bit.

The new content of the memory is computed offline in $O(\log N)$ time, where N is the number of prefixes. However, it is not necessary to download a new forwarding table for every route update. Route updates can be frequent, but routing protocols need time in the order of minutes to converge. Thus, the offline update computation can also be done at the control plane.

When a prefix update is initiated, the memory content of the write bubble table in each stage is updated (using the architecture shown in Fig. 12), and a write bubble is inserted into the pipeline. Each write bubble is assigned an ID. There is one write bubble table in each stage. The table stores the updated information associated with the write bubble ID. When it arrives at the stage prior to the stage to be updated, the write bubble uses its ID to look up the WBT. If the write enable bit is set, the write bubble will use the new content to update the memory location in the next stage. Due to the dual-ported memory, up to 2 nodes can be simultaneously updated at each stage. This updating mechanism supports non-blocking prefix updates at system speed.

For new-route update, if the structure of the BST is changed, the BST needs to be rebuilt, and the entire memory content of each pipeline stage needs to be reloaded. If the structure of the BST is not changed, the above update mechanism can still be used.

6.5 Update in 2-3-tree

The bottom-up approach is used to update the 2-3-tree. For the insertion, the first task is to find the (non-leaf) node that will be the parent p of the newly inserted node n. There are 2 cases: (1) parent node p has only 2 children and (2) parent node p has 3 children.

In this first case, n is inserted as the appropriate child of p; p becomes a 3-node. A new 2-node needs to be added for p, and a new 2-node is allocated for n in the next level. In the second case, n is still inserted as the appropriate child of p. However, p now has 4 children, which violates the property of a 2-3-tree. Hence, an internal node m is created with p's two rightmost children. m is then added as the appropriate new child of p's parent (i.e., add m just to the right of p). If p's parent had only 2 children, insertion is done. Otherwise,

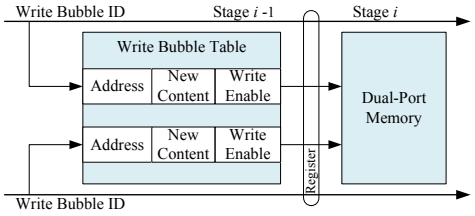


Fig. 13: Route update using dual write-bubbles

new nodes are created recursively up the tree. If the root is given 4 children, then a new node m is created as above, and a new root node is created with p and m as its children. Therefore, in the worst case, the insertion causes changes in *at most two nodes* at each level of the tree. The deletion process is similar, with merging instead of splitting.

Update in 2-3-tree can be performed using the similar update bubble mechanism as described above. When a prefix update is initiated, the memory content of the write bubble table in each stage is updated, and a write bubble is inserted into the pipeline. Due to the dual-ported memory, up to 2 nodes can be simultaneously updated at each stage. Since at most two nodes are modified in each level of the tree, one update bubble is enough to make all the necessary changes. When the bubble moves to the next level (stage), the tree up to that level is fully updated, and the subsequent lookup can be performed properly. Hence, this updating mechanism supports *single-cycle* non-blocking prefix updates at system speed.

6.6 Scalability

BST-based architecture: The use of the complete BST data structure leads to a linear storage complexity in our design, as each node contains exactly one prefix. The height of a complete BST is $(1 + \lfloor \log_2 N \rfloor)$, where N is the number of nodes. Since each level of the BST is mapped to a pipeline stage, the height of the BST determines the number of stages. Our proposed architecture is simple, and is expected to utilize a small amount of logic resource. Therefore, the major constraint that dictates the number of pipeline stages, and in turn the size of supported routing table, is the amount of on-chip memory. The scalability of our BST architecture relies on the close relationship between the size of the routing tables and the number of required pipeline stages. If the number of prefixes increases and the last level is full, then extra pipeline stages are added. For each additional stage, the size of the supported routing table is doubled.

2-3**-tree-based architecture**: Similarly to the BST approach, the use of the 2-3 tree leads to a linear storage complexity in our design, as each *effective* 2-node contains *exactly* one prefix. Each level of the 2-3-tree is also mapped to a pipeline stage. Thus, the height of the tree determines the number of pipeline stages. The scalability of this architecture is similar to that of the BST design.

7 Hardware Implementations

7.1 Application-specific integrated circuit (ASIC)

Implementation in ASIC is quite simple. If the size of the target routing table is known, then the number of pipeline stages and the memory size in each stage are deterministic. Hence, an ASIC chip can be fabricated and is ready to be used. However, since the ASIC chip is not reconfigurable, future expansion of routing tables must be taken into consideration when designing the chip.

Due to the simple architectures, our designs are memory-bound. Therefore, the amount of memory requirement determines the clock rate of the chip and is analyzed in detail. In the BST design, a prefix in each group is represented by one node of the BST. Let N_i be the number of prefixes in group G_i. The total number of nodes in k BSTs (N_S) is calculated using (4). Recall that a 3-node is converted to two 2-nodes in the 2-3-tree. Due to this node-splitting scheme, the number of 2-nodes is exactly equal to the number of prefixes in each group. Thus, the total number of nodes in the 2-3-tree can also be calculated using (4).

As mentioned earlier, each node of the BST contains 3 fields: (1) a padded prefix (L bits), (2) the prefix length ($L_P = \log L$ bits), and (3) the corresponding *next hop index* (L_{NHI} bits). In case of the 2-3-tree, in addition to these 3 data fields, there are 2 extra fields: left and right child pointers (L_{Ptr} bits for each). Let M_1 and M_2 be the total memory requirement of the BST and 2-3-tree designs. M_1 and M_2 are computed using (5) and (6). In reality, nodes in the last level of the 2-3-tree need not contain any pointer as they have no children. Additionally, the number of leaf-nodes is at least half the total number of nodes of the 2-3-tree. Therefore, the total memory requirement of the 2-3-tree can be rewritten as in (7).

$$N_S = \sum_{i=1}^{k} N_i \tag{4}$$

$$M_1 = \sum_{i=1}^{k} N_i L_i + N_S(L_P + L_{NHI}) \tag{5}$$

$$M_2 = \sum_{i=1}^{k} N_i L_i + N_S(L_P + L_{NHI} + 2L_{Ptr}) \tag{6}$$

In these above equations, $L_P = \log L = 5$ (for IPv4) and $L_P = \log L = 7$ (for IPv6). For the sake of simplicity, we can assign $L_i = L, L_{NHI} = 5, L_{Ptr} = 16$. (5) and (7) can be simplified for both IPv4 and IPv6, as shown in (8)-(9). Note that the unit of memory consumption is in bits, unless otherwise stated.

$$M_2 = \sum_{i=1}^{k} N_i L_i + N_S(L_P + L_{NHI} + L_{Ptr}) \tag{7}$$

$$M_1^{IPv4} = N_S \times 42, \qquad M_1^{IPv6} = N_S \times 140 \tag{8}$$

$$M_2^{IPv4} = N_S \times 58, \qquad M_2^{IPv6} = N_S \times 156 \tag{9}$$

TABLE 4: Number of prefixes supported by a state-of-the-art FPGA device with 36 Mb of on-chip memory (values are derived using (8) and (9))

Architecture	BRAM		BRAM+SRAM	
	IPv4	IPv6	IPv4	IPv6
BST architecture	857K	257K	16M	8M
2-3-tree architecture	620K	230K	16M	4M

7.2 Field-programmable gate array (FPGA)

Implementation on FPGA requires more attention. Since FPGA devices have limited amount of on-chip memory, the size of the supported routing table is bounded. The maximum size solely depends on the size of available on-chip memory. The number of IPv4 and IPv6 prefixes that can be supported by a state-of-the-art FPGA device with 36 Mb of on-chip memory (e.g. Xilinx Virtex6), for both BST and 2-3-tree *without* using external SRAM, are shown in Table 4.

Due to the limited amount of on-chip memory, it is highly desirable that external SRAMs can be used. In our design, external SRAMs can be utilized to handle larger routing tables, by moving the last stages of the pipelines onto external SRAMs. However, due to the limitation on the number of I/O pins of these devices, only a certain number of stages can be fitted on SRAM. To avoid reprogramming the FPGA, we can allocate the maximum possible number of pipeline stages and use only what we need. The only drawback is that this approach introduces more latency (by the extra number of pipeline stages).

Currently, SRAM is available in $2-36$ Mb chips [25], with data widths of 18, 32, or 36 bits, and a maximum access frequency of over 500MHz. Each stage uses dual-ported memory, which requires two address ports and two data ports. The address width is 16 bits, and the data width is equal to the size of a node. Hence, in the BST architecture, each external stage requires ≈ 116 and ≈ 312 I/O pins for IPv4 and IPv6, respectively. In the 2-3-tree architecture, each external stage requires ≈ 148 and ≈ 344 I/O pins for IPv4 and IPv6, respectively. Note that Quad Data Rate (QDR) SRAM can also be used in place of SRAM to provide higher chip-density and access bandwidth.

The largest Virtex package, which has 1517 I/O pins, can interface with up to 6 banks of dual-ported SRAMs for IPv4, and up to 3 banks for IPv6. For each additional pipeline stage, the size of the supported routing table at least doubles. Moreover, since the access frequency of SRAM is twice that of our target frequency (200 MHz), and the operating frequency of the memory controller is at least that of the target frequency [35], the use of external SRAM will not adversely affect the throughput of our design. The number of IPv4 and IPv6 prefixes that can be supported for both BST and 2-3-tree using combination of BRAM+SRAM is depicted in Table 4.

DRAM can also be used in our design. However, employing DRAM in our design requires some modifications to the architecture. Due to its structural simplicity, DRAM has very high density and access bandwidth. The major drawback is its high access latency. Therefore, the design needs to have enough memory requests to DRAM in order to hide this expensive latency. One possible solution is to have multiple pipelines sharing the same DRAM module. Consequently, these pipelines must stall when waiting for the requested data to come back from the DRAM module. The solution for this issue deserves a topic by itself; and thus, is beyond the scope of this paper.

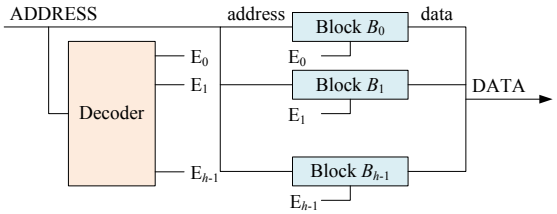


Fig. 14: Memory construction using primitive block B

8 Power Optimization

8.1 Memory Activation

Assume that the memory is constructed using primitive building blocks B, as shown in Fig. 14. Further assume that each block can store N_B tree nodes. As previously stated, in our design, the power dissipation in the memory dominates that in the logic. Thus, reducing power consumption in memory contributes to a large reduction in the total power consumption. Note that a SRAM block consumes power as soon as it is enabled and clocked, even if no input signals are changing. In each stage of the pipeline, more than one node is stored. However, only one node is accessed per clock cycle. This observation suggests a way to reduce memory power consumption by turning off the memory blocks that are not being accessed. We can easily achieve this by using a decoder to manually control each individual SRAM block. Let N be the number of nodes in the tree and h be the number of levels of the tree: $h = \lceil \log_2 N \rceil$ for a complete BST and $h \leq \lceil \log_2 N \rceil$ for a 2-3-tree. The power saving factor can be calculated as $S_P = \frac{N}{(h \times N_B)}$. It is obvious that S_P depends on the number of nodes per block N_B, or the granularity of B. When the size of B decreases, N_B decreases, and S_P increases. However, small values of B also increase the routing complication from the decoder to each memory block B, and in turn, adversely affect the clock rate of the design. Therefore, the size of B should be chosen to give a balanced design.

8.2 Cross-Pipeline Optimization

We can further reduce power consumption across the pipelines using the longest prefix matching property. If a match with length l has been found, there is no need to find matches with lengths less than l. Hence, in Fig. 9, if pipeline i has matched a prefix at stage j, the memory accesses in the subsequent stages in pipelines 1 to i are

not necessary; and therefore, can be turned off to save power. To support this mechanism, the match signal from stage j of pipeline i is forwarded to stage $j+1$ of pipelines 1 to $i-1$. All the incoming match signals to a stage are ORed together to generate a single signal. Each pipeline stage checks if the ORed match signal is set before accessing the local memory. In the worst case scenario, all the pipelines must traverse to the last stages to find a match. In this case, the only power saving that can be achieved comes from the memory activation technique.

9 Performance Evaluations

9.1 Experimental Setup

Fourteen experimental IPv4 core routing tables were collected from Project - RIS [22] on $06/03/2010$. These core routing tables were used to evaluate our algorithm for a real networking environment. Additionally, synthetic IPv4 routing tables generated using FRuG [13] were also employed as we did not have access to any real provider edge routing tables. FRuG takes a seed routing table and generates a synthetic table with the same statistic as that of the seed. From these core and edge routing tables, we generated the corresponding IPv6 edge routing tables using the same method as in [33]. The IPv4-to-IPv6 prefix mapping is *one-to-one*. Hence, the number of prefixes in an IPv4-IPv6 table pair is identical. The numbers of prefixes of the experimental routing tables are shown in Table 5 and 6.

9.2 Memory Efficiency

In general, the value of k is chosen by design engineers, based on the resource constraint. For the sake of simplicity, the prefix partitioning algorithm is evaluated with $k = 6$. The memory efficiency is expressed as the amount of memory (in bytes) required to store one byte of prefix. The results are shown in Table 5 and 6 for BST architecture and 2-3-tree architecture, respectively. In these tables, the memory efficiency for architecture using the uni-bit trie data structure (UBT) is also reported.

The results are consistent across different routing tables. It is clear that the BST data structure is slightly more memory efficient than the 2-3-tree data structure. However, the difference is not very significant if we consider the benefit of incremental update capability offered by the 2-3-tree architecture.

In IPv4, our partitioning algorithm is at least $2\times$ more memory efficient than the simple leaf-pushing approach [32] and the standard uni-bit trie approach. In IPv6, the results are drastically better. The memory efficiency of our algorithm is $50\times$ and $25\times$ than that of the simple leaf-pushing approach and the uni-bit-trie approach, respectively. Moreover, the memory efficiency of our design does not vary for IPv4 and IPv6 routing tables. It shows that the proposed algorithm and data structures scale very well to support the IPv6 standard with much longer prefix lengths. The memory consumption depends solely on the number of prefixes.

9.3 Throughput

ASIC Implementation: The largest backbone IPv4/v6 routing tables (rrc00) consisting of 330K prefixes were used to construct the evaluated architectures. The memory size of the largest stage is at most 10 Mbit and 26 Mbit for IPv4 and IPv6, respectively. Since the speed of the design depends mainly on the memory speed and not the logic, the access time of the largest memory block determines the overall speed. CACTI 5.3 [7] was used to estimate the memory access time on ASIC. A 10 Mbit dual-ported SRAM using 45 nm technology needs 2.039 ns to access, and a 26 Mbit dual-ported SRAM requires 3.103 ns. The maximum clock rate of the above architectures in ASIC implementation can be 490 MHz and 322 MHz for IPv4 and IPv6, respectively. These clock rates translate to 980 and 644 million lookups per second (MLPS). The results surpass the worst-case 150 MLPS required by the standardized 100GbE line cards [30].

FPGA Implementation: The same architectures were implemented in Verilog, using Synplify Pro $9.6.2$ and Xilinx ISE 12.4, with Virtex-6 XC6VSX475T as the target. The chip contains 74400 slices and 38304 Kbit of on-chip BRAM (1064 36-Kb blocks). The place and route results are collected in Table 7. Using the BST architecture, the IPv4 architecture fits in the on-chip BRAM of the FPGA. However, the IPv6 architecture requires some stages moved onto external SRAMs. Due to the relatively small size of the routing table, only one external stage is required. With this configuration, the IPv6 BST architecture employs only 3.2 Mbit of SRAM to support a core IPv6 routing table with the same number of prefixes (330K). Using dual-ported memory, the design can support 410 and 390 MLPS for IPv4 and IPv6, respectively.

Similarly, the IPv4 2-3-tree architecture fits in the on-chip BRAM of the FPGA, but the IPv6 architecture requires 2 external stages, which consumes 32.5 Mbit of SRAM to support the same IPv6 table. Using dual-ported memory, the design can support 401 and 373 MLPS for IPv4 and IPv6, respectively.

9.4 Performance Comparison

The memory efficiency and update complexity of the proposed algorithm and architecture are compared with those of the state-of-the-art approaches. These candidates are: the Distributed and Load Balanced Bloom Filters (DLB-BF) from [28], the Controlled Prefix Expansion (CPE) from [32], the B-tree Dynamic Router (BDR) engine from [24], and the Range Trie (RT) from [31]. The comparison results are shown in Table 8. In this table, N is the number of prefixes and L is the maximum prefix length.

With respect to the memory efficiency, our approach compares favorably with the state-of-the art techniques.

TABLE 5: Memory efficiency (Byte memory/Byte prefix) of the Uni-bit trie (UBT), Leaf-Pushing (LP), and Prefix Partitioning (DPP) algorithms for different experimental IPv4/v6 routing tables using BST data structure and architecture

Core routing tables								Edge routing tables							
Table	# prefixes	IPv4			IPv6			Table	# prefixes	IPv4			IPv6		
		UBT	LP	DPP	UBT	LP	DPP			UBT	LP	DPP	UBT	LP	DPP
rrc00	332118	2.88	2.08	0.98	27.14	49.15	0.88	rrc00_e	95048	4.13	3.69	0.99	90.12	47.31	0.90
rrc01	324172	2.88	2.08	1.00	27.31	49.47	0.90	rrc01_e	98390	5.83	5.51	0.96	88.52	51.13	0.92
rrc03	321617	2.87	2.07	1.01	27.12	49.12	0.91	rrc03_e	90867	4.13	3.70	0.99	87.23	47.25	0.94
rrc04	347232	2.88	2.07	0.94	27.61	50.04	0.84	rrc04_e	95358	3.35	2.88	1.04	95.86	45.21	0.89
rrc05	322997	2.87	2.07	1.01	27.10	49.07	0.91	rrc05_e	98305	4.42	4.01	0.99	87.52	48.47	0.91
rrc06	321577	2.88	2.08	1.01	27.16	49.20	0.91	rrc06_e	97410	5.73	5.39	0.96	87.35	50.74	0.91
rrc07	322557	2.88	2.07	1.01	27.19	49.26	0.91	rrc07_e	95007	4.98	4.59	0.97	87.71	49.24	0.89
rrc10	319952	2.87	2.06	1.02	27.12	49.12	0.92	rrc10_e	97376	4.14	3.70	0.99	86.77	47.33	0.89
rrc11	323668	2.88	2.07	1.00	27.19	49.26	0.91	rrc11_e	91031	4.95	4.56	0.97	88.02	49.14	0.94
rrc12	320015	2.88	2.07	1.02	27.12	49.13	0.92	rrc12_e	97386	3.38	2.91	1.04	86.80	45.34	0.94
rrc13	335154	2.89	2.09	0.97	27.29	49.44	0.87	rrc13_e	95163	4.96	4.57	0.97	91.47	49.16	0.94
rrc14	325797	2.87	2.07	1.00	27.20	49.27	0.90	rrc14_e	95032	4.20	3.77	0.99	88.62	47.61	0.90
rrc15	323986	2.87	2.07	1.00	27.14	49.16	0.90	rrc15_e	91019	4.14	3.71	0.99	87.93	47.34	0.90
rrc16	328295	2.87	2.07	0.99	27.09	49.07	0.89	rrc16_e	94780	3.38	2.92	1.04	88.95	45.36	0.91
Average		2.88	2.07	1.00	27.20	49.27	0.90	Average		4.41	3.99	0.99	88.78	47.90	0.91

TABLE 6: Memory efficiency (Byte memory/ Byte prefix) of the Uni-bit trie (UBT), Leaf-Pushing (LP) and Prefix Partitioning (DPP) algorithms for different experimental IPv4/v6 routing tables using 2-3-tree data structure and architecture

Core routing tables								Edge routing tables							
Table	# prefixes	IPv4			IPv6			Table	# prefixes	IPv4			IPv6		
		UBT	LP	DPP	UBT	LP	DPP			UBT	LP	DPP	UBT	LP	DPP
rrc00	332118	2.88	2.87	1.09	27.14	54.77	1.00	rrc00_e	95048	4.13	5.10	1.14	90.12	52.72	1.03
rrc01	324172	2.88	2.87	1.11	27.31	55.13	1.02	rrc01_e	98390	5.83	7.61	1.10	88.52	56.97	1.02
rrc03	321617	2.87	2.86	1.12	27.12	54.74	1.03	rrc03_e	90867	4.13	5.11	1.14	87.23	52.65	1.01
rrc04	347232	2.88	2.86	1.04	27.61	55.76	0.95	rrc04_e	95358	3.35	3.97	1.19	95.86	50.38	1.03
rrc05	322997	2.87	2.85	1.12	27.10	54.68	1.02	rrc05_e	98305	4.42	5.54	1.14	87.52	54.01	1.03
rrc06	321577	2.88	2.87	1.12	27.16	54.82	1.03	rrc06_e	97410	5.73	7.45	1.10	87.35	56.54	1.03
rrc07	322557	2.88	2.86	1.12	27.19	54.89	1.02	rrc07_e	95007	4.98	6.33	1.11	87.71	54.87	1.05
rrc10	319952	2.87	2.85	1.13	27.12	54.73	1.03	rrc10_e	97376	4.14	5.11	1.14	86.77	52.74	1.02
rrc11	323668	2.88	2.87	1.12	27.19	54.89	1.02	rrc11_e	91031	4.95	6.30	1.11	88.02	54.75	1.04
rrc12	320015	2.88	2.86	1.13	27.12	54.74	1.03	rrc12_e	97386	3.38	4.02	1.19	86.80	50.52	1.05
rrc13	335154	2.89	2.89	1.08	27.29	55.10	0.99	rrc13_e	95163	4.96	6.31	1.11	91.47	54.78	1.05
rrc14	325797	2.87	2.86	1.11	27.20	54.91	1.01	rrc14_e	95032	4.20	5.21	1.14	88.62	53.05	1.00
rrc15	323986	2.87	2.86	1.11	27.14	54.78	1.02	rrc15_e	91019	4.14	5.12	1.14	87.93	52.75	1.02
rrc16	328295	2.87	2.86	1.10	27.09	54.68	1.01	rrc16_e	94780	3.38	4.03	1.19	88.95	50.54	1.00
Average		2.88	2.86	1.11	27.20	54.90	1.01	Average		4.41	5.51	1.14	88.78	53.38	1.02

TABLE 7: FPGA implementation results

Architecture	Clock period (ns)	Slices	BRAM (36-Kb block)	SRAM (Mbit)
BST-IPv4	4.875	9054	402	0
BST-IPv6	5.125	14096	1025	3.2
2-3-tree-IPv4	4.986	10453	575	0
2-3-tree-IPv6	5.355	15358	580	32.5

TABLE 8: Comparison results

Architecture	Memory efficiency		Update complexity
	IPv4	IPv6	
DLB-BF	2.67	4.00	$O(N)$ in the worst case, 1 cycle otherwise
CPE	≥ 8.76	Not reported	$O(L)$
BDR	≥ 8.81	Not reported	$O(\log N)$
RT	$[1, 2]$	$[1, 2]$	$O(\log N)$
Our design	$[1, 1.11]$	$[0.9, 1.02]$	1 cycle

Additionally, our 2-3-tree-based design outperforms other approaches with regard to the incremental update capability. Only the BLP-BF architecture can have the same update speed, but in the worst case, when the hash tables change (which is more likely), the entire memory needs to be reloaded.

We cannot directly compare the throughput of these designs due to the lack of common hardware implementation platform and technology. However, with a simple and linear architecture, our design should have a comparable throughput with that of these existing approaches.

10 Concluding Remarks

In this paper, we have described an algorithm that can partition a given routing table into groups of disjoint prefixes. By exploiting the intrinsic distribution of the IP routing tables, the algorithm *minimizes* the total amount of memory required to store these tables. We also presented 2 data structures and their supporting architecture to achieve a *very high memory efficiency* of

1 byte of memory for each byte of prefix. These architectures also support *single-cycle* update. Furthermore, the proposed algorithm and architectures scale well to support very large forwarding tables. With these advantages, our algorithm and design can be used to improve the performance (throughput and memory efficiency) of IPv4/v6 lookup to satisfy the following criteria: (1) fast internet link rates up to and beyond 100 Gbps at core routers, (2) increase in routing table size at the rate of 25-50K prefixes per year, (3) increase in the length of the prefixes from 32 to 128 bits in IPv6, (4) compact memory footprint that can fit in the on-chip caches of multi-core and network processors, and (5) reduction in per-virtual-router storage memory of network virtual routers.

REFERENCES

[1] 2-3 Tree. Online. http://en.wikipedia.org/wiki/2-3_tree.

[2] F. Baboescu, S. Rajgopal, L. Huang, and N. Richardson. Hardware implementation of a tree based IP lookup algorithm for oc-768 and beyond. In *Proc. DesignCon '05*, pages 290–294, 2005.

[3] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *Proc. ISCA '05*, pages 123–133, 2005.

[4] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *Proc. INFOCOM '03*, pages 64–74, 2003.

[5] M. Behdadfar, H. Saidi, H. Alaei, and B. Samari. Scalar prefix search - a new route lookup algorithm for next generation internet. In *Proc. INFOCOM '09*, 2009.

[6] BST. Online. http://en.wikipedia.org/wiki/Binary_search_tree.

[7] CACTI Tool. Online. http://quid.hpl.hp.com:9081/cacti/.

[8] H. J. Chao and B. Liu. *High performance switches and routers*. JohnWiley & Sons, Inc., Hoboken, NJ, USA, 2007.

[9] CompleteBST. Online. http://xlinux.nist.gov/dads//HTML/completeBinaryTree.html.

[10] Dynamic Programming. Online. http://en.wikipedia.org/wiki/Dynamic _programming.

[11] H. Fadishei, M. S. Zamani, and M. Sabaei. A novel reconfigurable hardware architecture for IP address lookup. In *Proc. ANCS '05*, pages 81–90, 2005.

[12] J. Fu and J. Rexford. Efficient ip-address lookup with a shared forwarding table for multiple virtual routers. In *CoNEXT '08: Proceedings of the 2008 ACM CoNEXT Conference*, pages 1–12, 2008.

[13] T. Ganegedara, W. Jiang, and V. Prasanna. Frug: A benchmark for packet forwarding in future networks. In *IPCCC '10: Proceedings of IEEE IPCCC 2010*, 2010.

[14] W. Jiang and V. K. Prasanna. A memory-balanced linear pipeline architecture for trie-based ip lookup. In *Proc. HOTI '07*, pages 83–90, 2007.

[15] A. Kennedy, X. Wang, Z. Liu, and B. Liu. Low power architecture for high speed packet classification. In *Proc. ANCS*, 2008.

[16] S. Kumar, M. Becchi, P. Crowley, and J. Turner. CAMP: fast and efficient IP lookup architecture. In *Proc. ANCS '06*, pages 51–60, 2006.

[17] H. Le, W. Jiang, and V. K. Prasanna. A sram-based architecture for trie-based ip lookup using fpga. In *Proc. FCCM '08*, 2008.

[18] H. Le and V. K. Prasanna. Scalable high throughput and power efficient ip-lookup on fpga. In *Proc. FCCM '09*, 2009.

[19] H. Lu and S. Sahni. A b-tree dynamic router-table design. *IEEE Trans. Comput.*, 54(7):813–824, 2005.

[20] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[21] L. Peng, W. Lu, and L. Duan. Power efficient ip lookup with supernode caching. *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pages 215–219, Nov. 2007.

[22] RIS RAW DATA. Online. http://data.ris.ripe.net.

[23] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network*, 15(2):8–23, 2001.

[24] S. Sahni and K. S. Kim. An o($\log n$) dynamic router-table design. *IEEE Trans. on Computers*, 53(3):351–363, 2004.

[25] SAMSUNG SRAMs [Online. Online. http://www.samsung.com.

[26] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani. Scalable, memory efficient, high-speed ip lookup algorithms. *IEEE/ACM Trans. Netw.*, 13(4):802–812, 2005.

[27] K. Sklower. A tree-based packet routing table for berkeley unix. In *Winter Usenix Conf.*, pages 93–99, 1991.

[28] H. Song, F. Hao, M. Kodialam, and T. Lakshman. Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *Proc. INFOCOM '09*, 2009.

[29] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman. Building scalable virtual routers with trie braiding. In *INFOCOM'10: Proceedings of the 29th conference on Information communications*, pages 1442–1450, Piscataway, NJ, USA, 2010. IEEE Press.

[30] H. Song, M. S. Kodialam, F. Hao, and T. V. Lakshman. Scalable ip lookups using shape graphs. In *Proc. ICNP '09*, 2009.

[31] I. Sourdis, G. Stefanakis, R. de Smet, and G. N. Gaydadjiev. Range tries for scalable address lookup. In *Proc. ANCS '09*, 2009.

[32] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Trans. Comput. Syst.*, 17:1–40, 1999.

[33] M. Wang, S. Deering, T. Hain, and L. Dunn. Non-random generator for ipv6 tables. In *HOTI '04: Proceedings of the High Performance Interconnects, 2004*, pages 35–40, 2004.

[34] P. Warkhede, S. Suri, and G. Varghese. Multiway range trees: scalable ip lookup with fast updates. *Comput. Netw.*, 44(3):289–303, 2004.

[35] Xilinx. Online. http://www.xilinx.com/support/documentation/application_notes/xapp802.pdf.

Hoang Le received his BS (2005) and MS (2007) in Computer Engineering from George Mason University, MS (2010) in Electrical Engineering, and Ph.D (2011) in Computer Engineering from the University of Southern California. His research interests include: Network security, High-speed routers, Network virtualization, Secure embedded systems, and Cryptographic hardware. He is a member of IEEE and ACM.

Viktor K. Prasanna is Charles Lee Powell Chair in Engineering in the Ming Hsieh Department of Electrical Engineering and Professor of Computer Science at the University of Southern California. His research interests include High Performance Computing, Parallel and Distributed Systems, Reconfigurable Computing, and Embedded Systems. He received his BS in Electronics Engineering from the Bangalore University, MS from the School of Automation, Indian Institute of Science and Ph.D in Computer Science from the Pennsylvania State University. He is the Executive Director of the USC-Infosys Center for Advanced Software Technologies (CAST) and is an Associate Director of the USC- Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). He also serves as the Director of the Center for Energy Informatics at USC. He served as the Editor-in-Chief of the IEEE Transactions on Computers during 2003-06. Currently, he is the Editor-in-Chief of the Journal of Parallel and Distributed Computing. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the Steering Co-Chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the Steering Chair of the IEEE International Conference on High Performance Computing (HiPC). Prasanna is a Fellow of the IEEE, the ACM and the American Association for Advancement of Science (AAAS). He is a recipient of the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University.