Dynamic Multiway Segment Tree for IP Lookups and the Fast Pipelined Search Engine

Yeim-Kuan Chang, *Member*, *IEEE Computer Society*, Yung-Chieh Lin, and Cheng-Chien Su

Abstract—A dynamic multiway segment tree (DMST) is proposed for IP lookups in this paper. DMST is designed for dynamic routing tables that can dynamically insert and delete prefixes. DMST is implemented as a B-tree that has all distinct end points of ranges as its keys. The complexities of search, insertion, deletion, and memory requirement are the same as the existing multiway range tree (MRT) and prefix in B-tree (PIBT) for prefixes. In addition, a pipelined DMST search engine is proposed to further speed up the search operations. The proposed pipelined DMST search engine uses off-chip SRAMs instead of on-chip SRAMs because the capacity of the latter is too small to hold large routing tables and the cost of the latter is too expensive. By utilizing current FPGA and off-chip SRAM technologies, our proposed five-stage pipelined search engine can achieve the worst case throughputs of 33.3 and 41.7 million packets per second (Mpps) with 144-bit and 288-bit wide SRAM blocks, respectively. Furthermore, a straightforward extension of the pipelined search engine with multiple independent off-chip SRAMs can achieve the throughput of 200 Mpps which is equivalent to 102 Gbps for minimal Ethernet packets of size 64 bytes.

Index Terms—Segment tree, elementary interval, B-tree, pipeline, FPGA.

1 Introduction

The Internet applications such as World Wide Web (WWW) and P2P applications have generated tremendous network traffic on the Internet and hence consume a large percentage of the Internet bandwidth. If the Internet is able to continue supporting good quality of service, the next-generation IP routers have to provide faster packet forwarding rate and quicker adaptation to route changes. All tasks that have to be executed by the router after receiving a packet can be divided into time-critical (fast path) and non-time-critical (slow path) operations depending on the packet type and its frequency. Time-critical operations that are operated on majority of the packets must be implemented in a highly efficient and optimized manner to keep up with the high link speed and router bandwidth. Among all the tasks performed by the router [2], IP table lookups are the most time consuming. In table lookups, the destination addresses are looked up against a *forwarding table* by a *forwarding engine* that determines the next-hops in the network, where the packets should be sent.

Existing IP table lookup schemes can be broadly classified into two categories: static and dynamic. The static schemes are designed with the assumption that the forwarding table is not frequently updated. A forwarding table precomputation is typically needed in static schemes for improving lookup speed and reducing memory requirement. The disadvantage of static schemes is that when a single prefix is added or deleted, the entire forwarding table may need to be rebuilt. Rebuilding a forwarding table has a negative impact on the lookup performance of routers. On the other hand, in dynamic schemes, frequent insertions and deletions [15] are performed in real time, and thus forwarding table rebuilding is not required.

In this paper, we solve the IP table lookup problem by treating prefixes in the forwarding table as ranges. A range $[e, f]$ matches the destination address d iff $e \leq d \leq f$. The proposed data structure called *dynamic multiway segment tree* (*DMST*) is suitable for dynamic range insertions and deletions. DMST is a B-tree in which each node is augmented with a range set called *canonical set*. The detailed data structures of canonical set can be found in [5]. Although both the multiway range tree (MRT) [29] and the range in B-tree (RIBT) [20] also use B-trees, their structures have the following disadvantages:

1. The keys used to build the B-trees in MRT and RIBT are the traditional end points. For example, e and f are the keys for range $[e, f]$. However, we use $e - 1$ and f as the keys based on the minus-1 end point definition proposed in [5]. The minus-1 end point scheme uses fewer keys than the traditional end point scheme. Hence, the height of the B-tree in DMST is smaller than that in MRT and RIBT.
2. In MRT and RIBT, each key requires an additional equal list or heap to record which ranges start or terminate at the key. Equal lists determine whether or not a key needs to be removed from the B-tree after a range is deleted. The proposed DMST uses the concept of elementary intervals to build the B-tree and the proposed key deletion rule can determine if a key has to be removed without the need of equal lists or heaps. As a result, the DMST node structure is smaller than that in MRT and RIBT.

Consequently, even though the asymptotic complexities of performing dynamic search and update operations, and the asymptotic memory requirements are the same in all the B-tree-based schemes, our performance experiments using

• *The authors are with the Department of Computer Science and Information Engineering, National Cheng Kung University, No. 1, Ta-Hsueh Road, Tainan 701, Taiwan, R.O.C.*
E-mail: {ykchang, p7894110, p7894104}@mail.ncku.edu.tw.

Manuscript received 27 Dec. 2008; revised 5 Aug. 2009; accepted 16 Sept. 2009; published online 29 Sept. 2009.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-12-0638.
Digital Object Identifier no. 10.1109/TC.2009.153.

real IPv4 routing tables show that DMST is faster in terms of searches and updates and also consumes less memory than MRT and RIBT.

We also propose a pipelined search engine to further speed up the search operations. We choose off-chip SRAMs instead of on-chip SRAMs because the former is a lot cheaper than the latter [1], [10]. Furthermore, the storage capacity of off-chip SRAM can be as large as possible to meet our needs for storing a very large routing table. Two unique contributions of our paper are:

1. In theory, our DMST extends on the binary version of segment trees proposed in [5]. Since the height of multiway segment trees is smaller than that of binary segment trees, DMST achieves faster search speeds. Also, DMST is better than MRT and RIBT, the existing similar B-tree-based data structures for IP lookups.
2. In practice, our pipelined DMST search engine is designed for achieving a high throughput of up to 200 million packets per second by taking advantage of the fact that a search operation traverses less number of nodes in multiway segment trees than binary segment trees. The usage of off-chip SRAMs allows very large routing tables to be accommodated in the proposed pipelined search engine, which is not possible if on-chip memory is used.

The rest of the paper is organized as follows: Related works are discussed in Section 2. The preliminaries and design model are given in Section 3, and the detailed algorithms for DMST are presented in Section 4. The detailed architecture of the pipelined DMST search engine and its extension are proposed in Section 5. The results of the performance comparisons are given in Section 6, and finally, the concluding remarks are presented.

2 Related Works

Although research in IP lookup problem [6], [23] is conducted intensively in recent years, algorithms that balance among lookup speeds, memory requirement, update performance, and scalability are scarce. The existing schemes [9], [12], [16], [22], [28], [25], [13], [14] are mostly static and thus cannot afford frequent prefix insertions and deletions. The trie-based schemes [6], [23] like the binary trie, multibit trie, and Patricia trie do not use precomputation, and thus are suitable for dynamic forwarding tables. However, their lookup speeds degrade linearly with the address length and their memory consumption is large.

Sahni and Kim [24] developed a dynamic data structure called the collection of red-black trees (CRBT). The basic interval tree of CRBT is constructed from the traditional end points of all prefixes. CRBT supports search, insert, and delete operations in O(log N) time each for a routing table of N prefixes. Lu and Sahni proposed a dynamic scheme [18] based on an enhanced priority search tree [21] which arrives at O(log N) time complexity for search, insertion, and deletion. The experimental results in [18] showed that PST performs a little worse than CRBT in terms of search time. However, PST performs much better than CRBT in terms of insertion, deletion, and memory usage.

Lu and Sahni also developed an enhanced interval tree [3] which is called the binary tree on binary tree (BOB) [19] for dynamic routing tables. With real routing tables, BOB and prefix BOB perform the operations of insertion, deletion, and search in $O(logN)$ time. Also, the prefix BOB and the longest matching prefix BOB perform much better than PST in terms of search, insertion, deletion, and memory requirement.

A dynamic multiway fat inverted segment tree (FIS) is proposed in [11] for dynamic insertions and deletions of ranges. The search time of $o(\log_m N)$ can be achieved in a tree of degree m. In [29], a B-tree-based MRT is proposed to find the longest matching prefix in $o(\log_m N)$ time, and insert or delete a prefix in $o(m \log_m N)$ time. MRT is suitable for both prefixes and ranges. However, there are many duplicate end points stored in internal nodes, and a prefix may be stored in at most $m-1$ nodes in each level of B-tree. This drawback increases the update time and memory requirement. Another B-tree-based data structure called RIBT is proposed in [20] for solving this drawback by storing a range in only $O(1)$ B-tree nodes in each level of B-tree. The asymptotic complexity of PIBT to find the longest matching prefix is the same as MRT, and the measured time for the search operations is almost the same for RIBT and MRT using real routing tables. However, RIBT is more memory efficient than MRT by a constant factor. In addition to the B-tree-based algorithms mentioned above, Sun and Zhao [26] proposed some compression techniques that try to put as many nodes as possible into the B-tree nodes in order to fit entire routing table in the on-chip memory of a single chip. However, no efficient update algorithms can be supported because their compression techniques need precomputations. Also, the on-chip memory is expensive and the capacity of the on-chip memory is always limited by the chip area. The off-chip memory used in the proposed pipelined architecture is cheap and can be much larger than the on-chip memory for holding a very large routing table. Also, our proposed algorithm support dynamic updates.

Since DMST, MRT [29], and RIBT [20] all use multiway segment trees, the subtle differences among them are worth discussing. Their three major differences are 1) what are the keys, 2) how to store the keys, and 3) how to delete an end point in the B-tree which are specifically explained as follows:

The keys (i.e., end points) in MRT and RIBT are based on the traditional end point scheme in which the keys of range $R = [e, f]$ are defined as e and f, However, based on the minus-1 end point definition [5], $e-1$ and f are used as the keys in DMST. Let $E_{ef}(\mathcal{R})$ and $E_{e-1f}(\mathcal{R})$ be the sets of keys for a set R of N ranges based on the traditional end point and minus-1 end point, respectively. Obviously, both $|\mathrm{E}_{e,f}(\mathcal{R})|$ and $|\mathrm{E}_{e-1,f}(\mathcal{R})|$ are less than or equal to $2N$. $|\mathrm{E}_{e-1,f}(\mathcal{R})|$ is smaller than $|\mathrm{E}_{e,f}(\mathcal{R})|$ if some ranges in R are contiguous (i.e., the finish end point of a range is equal to the start end point of another range minus one). As shown in [5], the number of end points generated by the minus-1 end point scheme is about 69-73 percent of that by the traditional end point scheme. As a result, the memory requirement for DMST is less than that for MRT and RIBT.

With the difference in key definitions, MRT, RIBT, and DMST define the fundamentally different address coverage

(called *span* in MRT or *interval* in RIBT and DMST) for a key or node to facilitate the search process. Basically, MRT follows the span definition of the *binary range search* proposed in [16]. In MRT, the address span of a key v is defined as a half-open interval $(u, v]$, where u is the predecessor of v in the increasing order of keys. If addresses 0 and $2^W - 1$ in the W-bit address space are always included, the entire address space is the union of all address spans. Thus, as done in the binary range search [16], if we can precompute the highest priority range for each address span, a simple binary search can be applied to find which address span the address d belongs to, and then obtain the matched range for d. However, one problem is that the highest priority ranges for address span $(u, v-1]$ and singleton address v may be different. Thus, extra information for each end point is needed to determine the highest priority matched range for address d when d is equal to v or is larger than the predecessor of v. This is where the "$=$" and ">" ports of the binary range search [16] come from. In RIBT, the definition of interval is similar to the address span in MRT. The *key equal heap* which is the same as *key equal list* in MRT is needed for each key in RIBT. We will show later that the proposed DMST needs no key equal list or heap.

The key equal lists or heaps in MRT and RIBT can also be used to determine if a key can be removed from a node after a range is deleted. For example, if the ith equal heap of a node in RIBT is empty after a range is deleted, the ith key in that node can be deleted. In MRT, range $R = [e, f]$ is stored in the key equal list corresponding to key e in some leaf nodes. Since R is not stored in the key equal list of key f, a special counter is used to record how many ranges terminate at the key. Thus, when the counter associated with a key is zero, that key can be deleted. On the contrary, each node in DMST only stores a canonical set without the *equal* lists. Hence, a DMST node is smaller than an MRT or RIBT node. DMST uses the *key deletion rule* (see Definition 4 as described later) to determine if a key can be deleted or not.

After knowing which are the keys, it is necessary to decide how to store these keys in the B-tree. In MRT, all keys are stored in the leaf nodes at the bottom level of the B-tree. To build a multiway search tree, some of the keys in the leaf nodes are sent to upper levels. How many keys are sent up depends on the node order m. The process in which some of the keys in the leaf nodes are sent to upper levels will be performed repeatedly until the top level contains only one node. A key may be stored twice in MRT. Thus, the process of inserting or deleting an end point of the new range is complicated because at least two B-tree nodes are involved. In RIBT and the proposed DMST, each key is only stored in one node. Because of the different ways to store keys, the intervals in RIBT or address spans in MRT are defined differently.

3 Prerequisites and Model Design

The core part of the proposed DMST is based on the concepts of elementary intervals and the new end point definition proposed in [5]. By using the novel minus-1 end point definition, the DMST data structure is simpler than MRT and PIBT. To make this paper self-contained, we show the following two definitions proposed in [5]:

TABLE 1
An Example Routing Table of Nine 6-Bit Prefixes

ID	Prefix	Range	endpoints	
			start	finish
P1	000000/2	[0, 15]	-	15
P2	010000/2	[16, 31]	15	31
P3	000100/4	[4, 7]	3	7
P4	100000/1	[32, 63]	31	63
P5	010111/5	[22, 23]	21	23
P6	110000/2	[48, 63]	47	63
P7	110000/4	[48, 51]	47	51
P8	110111/6	[55, 55]	54	55
P9	100000/3	[32, 39]	31	39

Definition 1 (Elementary Interval). *Let the set of elementary intervals constructed from a set R consisting of NW-bit arbitrary ranges in the address space of 0 to $2^W - 1$ be $\mathcal{X} = \{X, X, = [e_i, f_i] \text{ for } i = 1 \text{ to } S\}$. $\mathcal{X}$ must satisfy*

1. $e_1 = 0$ *and* $f_S = 2^W - 1$,
2. $f_i = e_{i+j} - 1$ for $i = 1$ to $S - 1$,
3. *all addresses in X_i are covered by the same subset of $\mathcal{R}$ which is called the* range matching *set of X_i denoted by EI_i, and*
4. $EI_i \neq EI_{i+1}$ for $i = 1$ to $S - 1$.

Definition 2 (Minus-1 End Point Scheme). *The two end points of a range $[e, f]$ are defined to be $e - 1$ and f.*

Based on the minus-1 end point scheme, the set of end points built from the nine 6-bit ranges in Table 1 are {3, 7, 15, 21, 23, 31, 39, 47, 51, 54, 55}. By taking addresses 0 and 63 into consideration, 12 elementary intervals X_1-X_{12} can be constructed as shown in Fig. 1a. Every two consecutive elementary intervals cannot be covered by the same subset of the original ranges. For example, $\text{EI}_1 = \{\text{P1}\} \neq \text{EI}_2 = \{\text{P1}, \text{P3}\}$. Fig. 1b shows a possible multiway segment tree for the proposed DMST which will be explained later. The advantage of using multiway segment tree over the binary segment tree is that the number of multiway tree nodes traversed for a search is small. This advantage makes our pipelined search engine very suitable for the proposed DMST and leads to a high-throughput design.

Fig. 2 illustrates the design model for the proposed IP lookup algorithm. The protocol stack is divided into *data plane* and *control plane*. The control plane generally consists of a large number of sophisticated codes that implement the slow-path protocols such as IP routing protocol or higher layer protocols. The slow-path functions in control plane are managed by a standard RISC processor like the Xscale core in Intel IXP network processors. In this paper, we assume that the slow-path RISC processor mainly executes the route update operations based on the proposed dynamic multiway segment tree. The fast-path function is typically the IP, ATM, and similar protocols in layer two or three of the network

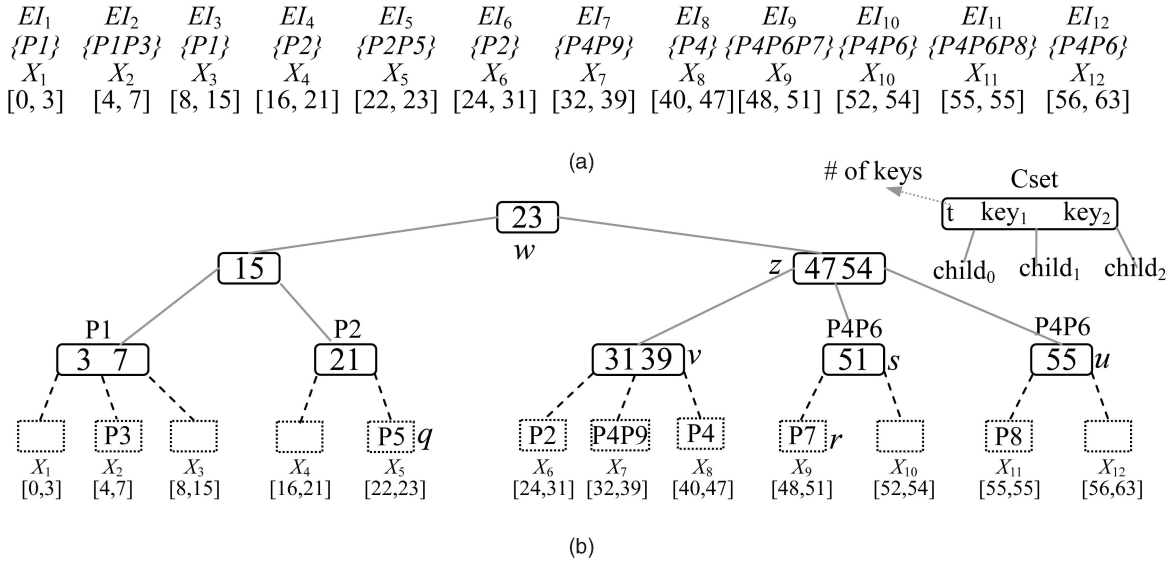

Fig. 1. (a) Elementary intervals and (b) a possible DMST built according to Table 1.

protocol stack. We also assume that the fast-path function implements the IP lookups by using dedicated FPGA-based search engine designed for the proposed DMST. The proposed pipelined search engine uses the off-chip instead of on-chip SRAMs because the memory storage capacity in current FPGA devices is too small for any existing IP lookup algorithm that can concurrently support dynamic routing table update operations and a large routing table of more than 100k routing entries. For example, the size of the SRAM memory provided by the current technology such as Xilinx Virtex-5 XC5VLX330T FPGA [30] is 11,664 Kb which is too small for large routing tables.

As shown in Fig. 2, the packets are received at the receiving unit and transmitted to the outside after determining the next port. If the packets are for updating routing table, they are sent to the slow-path RISC processor which will execute the proposed update operations and modify the contents of some memory blocks of the off-chip SRAM accordingly. If the packets are the usual Internet packets, they are sent to the proposed DMST search engine for finding the next port numbers and finally are passed to the transmitting unit.

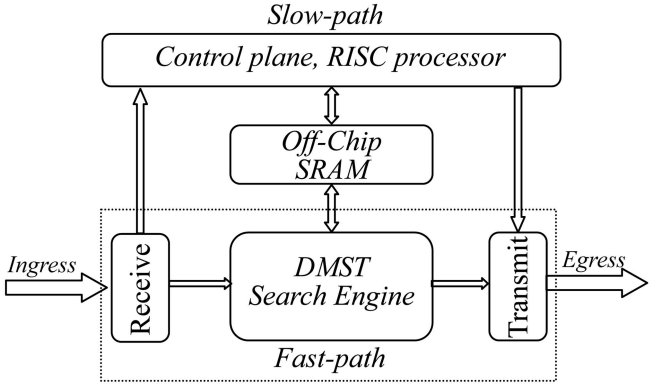

Fig. 2. Design model.

4 Proposed Dynamic Multiway Segment Tree (DMST)

In this paper, the elementary intervals are organized hierarchically as a segment tree called *DMST* which is efficient for dynamic insertions and deletions. DMST is implemented as a B-tree of order m. Fig. 1b shows a possible order-3 DMST for the prefixes in Table 1. Every node x is associated with an *interval* denoted by *intvl(x)* which is the union of elementary intervals in the subtree rooted at node x. Thus, the interval associated with the root node in DMST covers the entire address space $[0, 2^W - 1]$. Each node is also associated with a range set called *canonical set (Cset)*. Each end point is stored in exactly one node as its key.

Definition 3 (Range Allocation Rule). *Range R is stored in the canonical set of a node x if and only if intvl(x) is contained in R, but intvl(parent(x)) is not contained in R.*

Based on the range allocation rule, the range matching set of an elementary interval is equal to the union of the canonical sets traversed on the path from the root to the elementary interval. When range R1 is more specific than range R2, R1 must be stored in the lower level than R2. For example, P5 is more specific than P2 in Fig. 1b. Thus, node q storing P5 is in the lower level than node u which stores P2.

The data structure of an internal node consisting of t keys is formatted in a linear list as $[t, Cset, child_0, key_1, child_1, \ldots, key_t, child_t]$, where $child_i$ is a pointer to the ith subtree for $i = 1$ to t and Cset is the canonical set. Also, the t keys stored in an internal node satisfy the condition of $key_1 < key_2 < \cdots < key_t$. A leaf node only stores *Cset*. By saying "*insert a key ep and a pointer ptr as* key_i *and* child_i *in a node*," we mean $[key_i, child_i, \cdots, key_t, child_t]$ are shifted to the right by one key, *ep* and *ptr* are inserted as key_i and $child_i$, and t is incremented by one. In case of ambiguity, the filed fld of node x is denoted by $xfld$. If all keys are sorted in an increasing order, the successor and predecessor of a key k are denoted by *successor(k)* and $predecessor(k)$,

```
   Algorithm DMST_Search(root, d)
   {
01   x = root; k = 0;
02   while (x ≠ null) {
03        if (x.Cset ≠ ϕ) Cset[++k] = x.Cset;
04        if (x is a leaf node) break;
05        x.key_0 = predecessor(x.key_1); x.key_{x.t+1} = successor(x.key_{x.t});
06        Binary search on keys x.key_0 to x.key_{x.t+1};
07        if (x.key_{i-1} < d ≤ x.key_i) x = x.child_{i-1};
08   }
09   if (range set is conflict-free) return the highest priority range in Cset[k];
10   else return the highest priority range in Cset[1] to Cset[k];
   }
```

Fig. 3. The DMST search algorithm.

respectively. If $successor(k)$ does not exist, it is set to $2^W - 1$, and if $predecessor(k)$ does not exist, it is set to -1.

4.1 Search in DMST

Given a destination address d, the searching process finds the matching ranges that contain d. If each range is associated with a priority, the searching process finds the highest priority range among all matching ranges. In this paper, we use the traditional priority assignment rule to set the priority of a range as follows: Range R1 is assigned with a higher priority than range R2 if R1 is more specific than R2. The routing table from an IP router is a practical example. For IP routing tables, the longer the prefix, the higher is its priority. Therefore, the routing table lookups find the longest prefix among all matching prefixes of d.

Fig. 3 shows the proposed DMST search algorithm. A simple tree traversal in the *while loop* is first performed from the root to the leaf node that corresponds to the elementary interval containing d. While traversing the tree, all nonempty canonical sets encountered are stored in the array Cset $[1..k]$. If the range set is conflict free, the highest priority (the most specific) range must exist in the nonempty canonical set which was last visited. For example, when the DMST search algorithm is applied to Fig. 1b with $d = 48$, the nodes w, z, s, and r are traversed. The nonempty canonical sets are {P4, P6} and {P7}. As a result, the matching ranges of d are P4, P6, and P7, and the most specific matching range is P7.

Complexity. The complexity of the search algorithm in Fig. 3 for an order-m DMST of N arbitrary ranges depends on the data structure of the canonical set which is assumed to be a bitmap [5] in this paper. Currently, we make the following assumptions which are also used in insertion and deletion: *Each address is covered by at most maxR ranges, inserting/deleting a range into/from a canonical set of size Csize takes O(f(Csize)) time, and accessing the highest priority range takes O(g(Csize)) time.*

The binary search is used to determine where the given address d is located in a B-tree node. Thus, each iteration of the while loop takes $O(\log m)$ time to determine i such that $key_{i-1} < d \leq key_i$. If m is small and the whole node can fit into a cache block (e.g., L1 cache in a modern CPU), only a constant time is needed for the binary search in a node. The number of iterations is $O(\log_m N)$ (the height of the B-tree). After the tree traversal, the main task of the search algorithm is to find the highest priority range in a canonical set as shown in lines 9 and 10. If the range set is conflict free, only the last canonical set is searched, and thus the search complexity is $O(\log_m N \times \log m + g(\max R))$. Otherwise, the search complexity is $O(\log_m N \times \log m + \log_m N \times g(\max R))$. The number of nodes accessed is $o(\log_m N)$.

4.2 DMST Insertion Algorithm

We first propose an insertion algorithm that separately puts the two end points of the range and then the range itself into DMST. This approach was used in MRT [29] and RIBT [20]. Next, we will briefly describe the optimized insertion algorithm that combines these three steps into one. The optimized insertion algorithm avoids some redundant operations and thus performs better than the unoptimized insertion algorithm.

4.2.1 Insert an End Point

The proposed algorithm that inserts an end point *ep* into DMST is $DMST\_Insert\_EndPoint$ shown in Fig. 4. $DMST\_Insert\_EndPoint$ is an adaptation of the standard B-tree insertion algorithm [7] and is described as follows:

- Step 1: A tree traversal is performed to find the key *ep* in DMST. If *ep* is already in DMST, the search terminates at the node that contains the key *ep*. If *ep* is not in DMST, the algorithm terminates at a leaf node whose parent node will hold the key *ep*.
- Step 2: k is first decremented by one and x is set to $p[k]$. Let $p[k].key_0$ and $p[k].key_{xt+1}$ be the predecessor $(p[k].key_1) = s[k] - 1$ and the successor $(p[k].key_{x.t}) = f[k]$, respectively. Based on index $b[k]$, the new key *ep* will be inserted between $x.key_{b[k]-1}$ and $x.key_{b[k]}$. Since the original elementary interval $[x.key_{b[k]-1} + 1, x.key_{b[k]}]$ is partitioned into two intervals by *ep*, a new leaf node pointed to by y has to be created. Node y is a duplication of the leaf node pointed to by $x.child_{i-1}$. Keys *ep* and y are inserted as key_i and $child$, of node x, respectively, and $x.t$ is incremented by one.

```
Algorithm DMST_Insert_EndPoint(root, ep) // Assume tree is not empty
  {
////////////////////////////// Step 1: Traverse the tree for finding key ep ////////////////////////
01   Perform tree_traversal to find arrays p[], s[], f[], b[], and k, where
     p[i] for i = 1 to k are the nodes traversed, [s[i], f[i]] is the interval covered by p[i],
     b[i] is the index for node p[i] such that p[i].key_{b[i]-1} < ep < p[i].key_{b[i]};
02   if (p[k] ≠ leaf) return;
////////////////////////////// Step 2: insert key ep which is not in the tree ////////////////////////
03   k = k – 1; x = p[k]; i = b[k];
04   y = duplicate_a_leaf_node(x.child_{i-1}.Cset);
05   insert ep and y as x.key_i and x.child_i in node x, and x.t++;
////////////////////////// Step 3: node overflow, split x into two nodes, x and y ////////////////////
06   while (x.t = m){
07        g = ⌈m/2⌉; keyg = x.key_g;                                          } 3.1: Create a new node y
08        y = create_new_node();
09        move child_g, [key_{g+1}, child_{g+1}], ..., [key_m, child_m] in node x to node y;
10        y.Cset = x.Cset;
11        y.t = m – g; x.t = g – 1;
12        xSet = {R|R ∈ x.child_{g-1}.Cset and R covers [s[k], keyg]};        } 3.2: Adjust x.Cset
13        for (h= 0 ; h ≤ x.t ; h++) x.child_h.Cset = x.child_h.Cset – xSet;
14        x.Cset = x.Cset + xSet;
15        ySet = {R|R ∈ y.child_0.Cset and R covers [keyg + 1, f[k]]};        } 3.3: Adjust y.Cset
16        for (h = 0 ; h ≤ y.t ; h++) y.child_h.Cset = y.child_h.Cset – ySet;
17        y.Cset = y.Cset + ySet;
18        if (k = 1) {root = create_node(t = 1, child_0 = x, key_1 = ep, child_1 = y); break;}  } 3.4
19        k = k – 1; x = p[k]; j = b[k]; x.t++;
20        insert keyg and y as x.key_j and x.child_j in node x;   }
  }
```

Fig. 4. The algorithm that inserts a new end point into the DMST.

- Step 3: After inserting *ep*, if $x.t \leq m-1$, the insertion process is finished. Otherwise, node x has one key more than its capacity. Thus, in lines 7-11, node x needs to be split into two nodes denoted by x' and y, and the middle key key_g of x is sent up to $x's$ parent, where $g = \lceil m/2 \rceil$. Specifically, the keys of x to the left of key_g along with the associated child pointers remain in x, those to the right of key_g are placed into a new node y, and key_g and y are inserted into the parent of x. After node x is split, the canonical sets in x' and y need to be adjusted to account for the fact that $intvl(x')$ and $intvl(y)$ are not the same as $intvl(x)$. As stated in step 1, node x is pointed to by $p[k-1].child_{j-1}$ after the tree traversal, where $p[k-1]$ is the parent of x and $j = b[k-1]$.

Consider the example in Fig. 5. Range R1 that contains the interval $\text{intvl}(x') = [p.key_{j-1}+1, x.key_g]$ was stored in all canonical sets of the children of node x' before splitting. Thus, R1 needs to be removed from all these canonical sets of the children of x' and be inserted into $x'.Cset$. Similarly, range R2 that contains the interval $intv(y) = [x.key_g + 1, p.key_j]$ needs to be removed from all canonical sets of the children of y and be inserted into $y.Cset$. The above canonical set adjustments are shown in lines 12-17 of Fig. 4. Finally, key key_g and pointer y are inserted as key_j and $child_j$ in $p[k-1]$, respectively. Since node $p[k-1]$ gets one more key, the same split process may need to repeat at $p[k-1]$ if $p[k-1]$ were overflowed. Ultimately, the split process may reach the root of the tree. As in the regular B-tree, a new root node may need to be created and thus the height of the tree is increased by one, as shown in line 18 of Fig. 4.

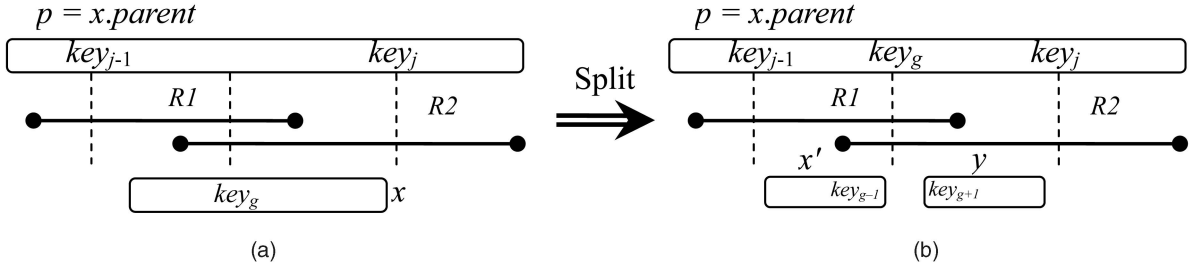

Fig. 5. Node splitting around $x.key_g$. (a) Before split. (b) After split.

```
Algorithm DMST_Insert_Range(root, R) // assume R=[e, f]
{
    ////////////////////////////////////////// Step 1 //////////////////////////////////////////
01  Find LCA node y and the interval [lb, ub] covered by y;
    ////////////////////////////////////////// Step 2 //////////////////////////////////////////
02  if ( [lb, ub] is contained in R ) { Add R in y.Cset; return;}
    ////////////////////////////////////////// Step 3 //////////////////////////////////////////
03  Set y.key_0 = lb – 1, y.key_{y.t+1} = ub
04  for (k = 1 to y.t+1)
05      if (R covers [y.key_{k–1}, y.key_k]) { Add R in y.child_{k–1}.Cset;}
    ////////////////////////////////////////// Step 4 //////////////////////////////////////////
06  if (y.key_{i–1}< e – 1 < y.key_i) { // i ∈ {1, …, y.t+1} and e – 1 ≠ any key in node y
07      x = y.child_{i–1};
08      while (x ≠ leaf node){
09          if (x.key_i = e – 1) { // i ∈ {1, …, x.t}
10              for (k = i to x.t) Add R in x.child_k.Cset;
11              break; }
12          if (x.key_{i–1} < e – 1 < x.key_i) {
13              for (k = i to x.t) Add R in x.child_k.Cset;
14              x = x.child_{i–1}; }
15      }
16  }
////////////////////////////////////////// Step 5 //////////////////////////////////////////
    if clause which is the same as lines 6-16 except
    1. 'e – 1' is replaced with 'f',
    2. line 10 is replaced with for (k=0 to i – 1) Add R in x.child_k.Cset;
    3. line 13 is replaced with for (k = 0 to i – 2) Add R in x.child_k.Cset
}
```

Fig. 6. The algorithm of the DMST range allocation rule.

4.2.2 Insert a Range

After inserting the two end points $e-1$ and f which are induced by the new range $R=[e,f]$, algorithm DMST_Insert_Range shown in Fig. 6 is then presented to insert range R based on the range allocation rule and described as follows:

- Step 1 (line 1): The lowest common ancestor (LCA) node y of R is found first. It is the lowest common ancestor node of keys $e-1$ and f in DMST.
- Step 2 (lines 2): If R contains the interval covered by the LCA node y of R, then R is added in $y.Cset$.
- Step 3 (lines 3-5): If R contains the interval associated with any of the children of LCA node y, R is added in that child's canonical set.
- Step 4 (lines 6-16): If $e-1$ is equal to any key in LCA node y, no further process is needed. If $y.key_{i-1} < e-1 < y.key_i$. the tree is traversed from node y to the node that contains the key $e-1$. At each node x traversed, R is added in the canonical sets of some of x's children as shown in line 10 or 13.
- Step 5: this step is similar to step 4.

For example, if range $\mathrm{P4}=[32,\ 63]$ in Table 1 is to be inserted, the end point 31 is first inserted in DMST as shown in Fig. 1b. Step 1 finds that the LCA node is z. Step 2 does nothing, and step 3 inserts P4 into $s.Cset$ and $u.$Cset because P4 contains $intvl(s)$ and $intvl(u)$ but not $intvl(z)$. In step 4, the leaf node v containing key 31 is reached and range P4 is inserted in $v.child_1.Cset$ and $v.child_2.Cset$ as shown in line 10 of Fig. 6.

4.2.3 Complexity

In the complexity analysis, the same assumptions as in DMST search algorithm are used.

1. *Algorithm DMST_Insert_EndPoint* in Fig. 4: Step 1 (lines 1 and 2) traverses $O(\log N)$ nodes. In each node traversed, the binary search on the keys takes $O(\log m)$ time. Thus, the time complexity of step 1 is $O(\log_m N \times \log m)$. Step 2 (lines 3-5) takes $O(m)$ time because inserting a key and a pointer in a node is needed. Each iteration of the while loop in step 3 is divided into four parts. Part 3.1 (lines 7-11) takes $O(m)$ time because a new node y consisting of $o(m)$ keys is created. In part 3.2 (lines 12-14), the canonical set in node x' is adjusted. With the assumption that the size of a canonical set is $O(\max R)$, $O(f(\max R))$ time is needed to remove a range from $x.child_{g-1}.Cset$, and the number of ranges removed from $x.child_{g-1}.Cset$ is $O(\max R)$. As a result, the complexity of part 3.2 is $O(\max R \times m \times f(\max R))$. Parts 3.3 and 3.4 are similar to parts 3.2 and 3.1, respectively. The number of iterations is $O(\log_m N)$ which is the height of the tree. Therefore, the total complexity is $O(\log_m N \times \max R \times m \times f(\max R))$.

2. *Algorithm DMST_Insert_Range* in Fig. 6: The complexity of step 1 is $O(\log_m \times \log m)$ by using a binary search on $O(m)$ keys in a node. Adding R in $y.Cset$ takes $O(f(\max R))$ time with the same assumption described in Section 4.1. Thus, the complexities of Step 2, 3, and 4 are $O(f(\max R))$, $o(m \times f(\max R))$, and $O(\log_m N \times m \times f(\max R))$, respectively. Step 5 is similar to step 4. Overall, the time complexity of inserting a range and the associated two end points is $O(\log_m N \times \max R \times m \times f(\max R))$.

4.2.4 Optimized DMST Insertion

In the optimized DMST insertion algorithm, range $R = [e, f]$ and the two associated keys $e-1$ and f are inserted in DMST at the same time. Since keys $e-1$ and f may not exist in DMST before R is inserted, a different definition for the LCA node of R is presented as follows: Let G be the set of existing keys contained in range $[e-1, f]$. If G is empty, the LCA node is defined to be the node into which keys $e-1$ and f will be inserted. If G is not empty, the LCA node is defined to be the lowest common ancestor of all nodes that contain any key in G. In practice, the LCA node is the first DMST node encountered when the B-tree is traversed from the root by using $[e-1, f]$. For example, to insert a range R = [34, 53] into the DMST in Fig. 1b, the LCA(R) is node z because z contains key 47 as covered by [33, 53]. However, if R = [34, 35], the LCA node of R is v.

The optimized DMST insertion algorithm to insert range R is briefly described as follows: The first step finds node y which is the LCA node of R. If R contains $intvl(y)$, R is immediately added in $y.Cset$ and insertion process stops. Now, consider the case in which end points $e-1$ and f of R are inserted between two consecutive keys in node y. If $y.key_{i-1} < e-1 < f < y.key_i$, keys $e-1$ and f, along with two newly created leaf nodes u and x, will be inserted in y. The canonical sets of nodes u and x are assigned with $y.child_{i-1}.Cset + R$ and $y.child_{i-1}.Cset$, respectively. The canonical set $y.child_{i-1}.Cset$ is duplicated in nodes u and x because the original elementary interval $[y.key_{i-1}+1, y.key_i]$ is divided into three smaller intervals by keys $e-1$ and f. The node splitting process is needed only if y is overflowed.

Finally, the case that interval $[e-1, f]$ covers at least one existing key in the tree has to be considered. First, R is directly inserted in $y.child_{k-1}$.Cset if R contains $[y.key_{k-1}+1, y.key_k]$, where y is the LCA node. Next, the insertion process is divided into two independent steps to insert $e-1$ and f, respectively. Since they are similar, only the process of inserting $e-1$ is described. The DMST tree is first traversed from the LCA node to a node that contains key $e-1$ or to a node that will hold the new key $e-1$. Let the interval covered by a traversed node x be $[lb, ub]$. Range R must contain $[e, ub]$. Therefore, R must be inserted in $x.child_k.Cset$ if $e-1 \leq x.key_k$ for $k = i$ to $x.t$. When key $e-1$ is not equal to any key in node x, the process will be repeated at the next level. After the last regular node is reached, if key $e-1$ does not still exist, a new leaf node is created and inserted. The time complexity of the optimized insertion algorithm is $O(\log_m N \times maxR \times m \times f(maxR))$, which is the same as that of the unoptimized version.

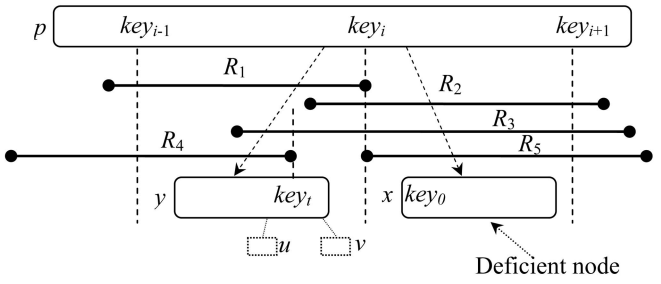

Fig. 7. The key borrowing operation from the left sibling.

4.3 DMST Deletion Algorithm

We only present the unoptimized deletion algorithm that divides the deletion process into three separate steps. The optimized deletion algorithm that combines these three steps into one single step can be developed easily by using the concept of the LCA node and is thus omitted. To delete a range $R = [e, f]$, R is first deleted from all canonical sets that contain R by using the reverse process of algorithm DMST_Insert_range() in Fig. 6. Then, we determine if key $e-1$ and f need to be removed from the tree based on the DMST *key deletion rule* described below.

Assume key $e-1$ or f of $R = [e, f]$ is key_i in node x that delimits the elementary intervals X_k and X_{k+1}. After R is removed from DMST, if EI_k and EI_{k+1} become the same, key_i must be removed from node x to satisfy Definition 2. As stated, EI_k is equal to the union of the canonical sets in the path from the root to the leaf corresponding to elementary interval X_k. Since the ranges in the canonical sets of node x's ancestors cover both X_k and X_{k+1} and thus belong to both EI_k and EI_{k+1}, they can be ignored when determining if $\mathrm{EI}_k = \mathrm{EI}_{k+1}$. As a result, EI_i and EI_{i+1} can be computed as follows (refer to Fig. 7):

$$\mathrm{EI}_k = x.\mathrm{child}_{i-1}.Cset \cup u_1.\mathrm{child}_{u1.t}.Cset \cup \ldots \cup u_n.\mathrm{child}_{un}.t.Cset, \quad (1)$$

$$\mathrm{EI}_{k+1} = x.child_i.Cset \cup v_1.\mathrm{child}_0.Cset \cup \ldots \cup v_n.\mathrm{child}_0.Cset, \quad (2)$$

where u_j and v_j for $j = 1$ to n are the descendent nodes of x in the paths from x to u_n and from x to v_n, and u_n and v_n are the nodes that contain keys $predecessor(x.key_i)$ and $successor(x.key_i)$, respectively.

Definition 4 (DMST Key Deletion Rule). *The* key_i *must be removed from node* x *if and only if* $\mathrm{EI}_k = \mathrm{EI}_{k+1}$, *where* EI_k *and* EI_{k+1} *are computed based on (1) and (2).*

For example, in Fig. 1b, the elementary interval X_6 is contained in intervals $intvl(w)$, $intvl(z)$, $intvl(\mathrm{v})$, and $intvl(\mathrm{v.child}_0)$. Thus, the range matching set EI_6 of X_6 is {P2} which is the union of $w.Cset$, $z.Cset$, $v.Cset$, and $v.\mathrm{child}_0.Cset$. Similarly, EI_5 is {P2, P5} which is the union of $w.Cset$, $y.Cset$, $u.Cset$, and $u.\mathrm{child}_1.Cset$. If $\mathrm{P5} = [22, 23]$ is deleted, key 23 in node w has to be deleted. Similarly, key 21 can also be deleted.

Like the B-tree deletion algorithm [7], deleting a key from DMST can be divided into two cases: deleting a key in a leaf node or in an interior (i.e., nonleaf) node. The detailed

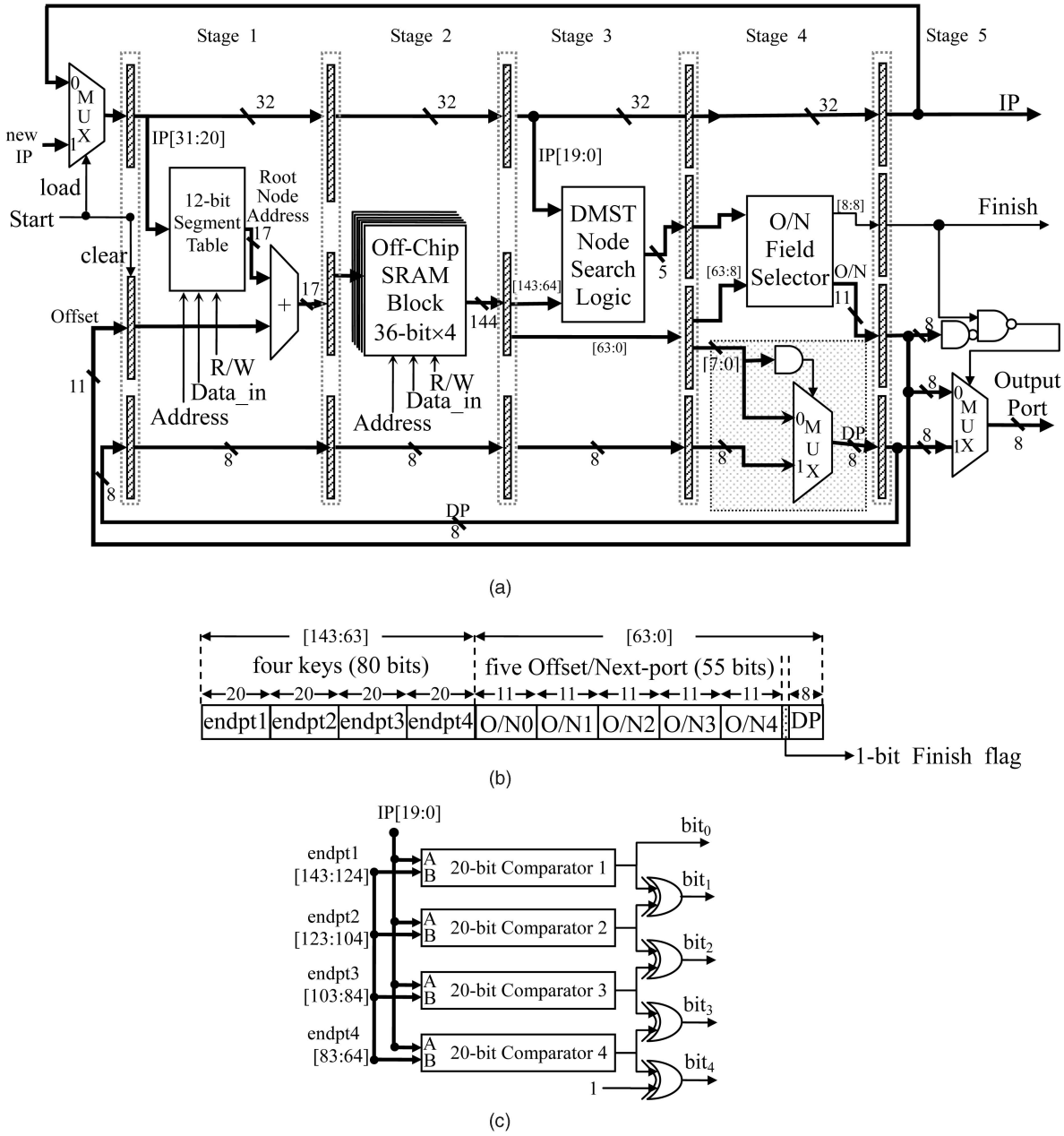

Fig. 8. The DMST search engine architecture. (a) The five-stage pipelined search engine, where DP and O/N stand for default port and offset/next port, respectively. (b) The 144-bit DMST node layout. (c) The DMST node search logic of stage 3, where the output of a 20-bit comparator is enabled if its inputs satisfy the condition of $A \leq B$.

key deletion algorithms are omitted in this paper because of the space limit.

5 The DMST Search Engine

In this section, we present the pipelined architecture of the search engine designed for the proposed DMST search algorithm. The pipelined architecture simulates all the operations needed for searching a B-tree node in DMST. If the DMST to be searched is an n-level B-tree, all the stages of the pipelined architecture will be circulated through n times to complete a search operation for an incoming IP address. Fig. 8a shows an example of a five-stage pipelined architecture that uses 144-bit nodes implemented by four 36-bit memory modules accessed in parallel. With the five-stage pipelined architecture, five different IP addresses can be searched in the search engine concurrently at each cycle and thus the lookup throughput is improved.

To efficiently utilize all the 144 bits in the 144-bit nodes, a 12-bit segmentation table [17] is used. As a result, up to 2^{12} independent multiway segment trees are built and 20 bits are needed to represent the end points. The number of 144-bit nodes in the maximal multiway segment tree is less than 2,048, as shown in column 4 of Table 3. Hence, 11 bits are needed for representing the offset address of a node in a multiway segment tree. The offset address is relative to the absolute address of the root node in the multiway segment tree. These two addresses (absolute address of root node and offset address of a node) are added together to get the absolute address of the node. Note that the offset addresses

correspond to the child pointers of the B-tree nodes in the software implementation of the DMST described in Section 4. In addition to the 20-bit end points and 11-bit offset addresses, each node also need 1 bit to indicate if it is leaf node (denoted by 1-bit *finish flag*) and an 8-bit default port number. The default port (denoted by *DP*) field is the next port number of the prefix that entirely covers the address space of the node. Therefore, as shown in Fig. 8b, all the 144 bits can be completely utilized to build a five-way DMST. The O/N field of a node in Fig. 8b indicates that the search outcome obtained from the node is either the offset address of the next-level node or the final next port number of the current search operation. The field t to record the number of keys in the node data structure previously described in Section 4 is not needed in the search engine. Therefore, if there are less than four keys in the node, the values of the unused key slots are set to the largest key value. An extreme case is as follows: Since a 12-bit segmentation table is used in DMST, there will be some segments containing no prefixes of length longer than 12. For this kind of segments, we store the largest address of the segment in all the four key slots. Also, the 1-bit finish flag is set to 1 and the next port number which must be O/N0 field is set to the port number of the longest prefix that completely covers the segment's address space. As a result, no additional logic is needed to process this exceptional case. The worst case search performance will be determined by the size of the maximal multiway segment tree. The 12-bit segmentation table is very small compared with the memory used for all the multiway segment trees. Similarly, the search engine of 288-bit nodes can be designed to build a 10-way DMST.

The functionalities of all five stages in the pipelined search engine consisting of 144-bit DMST nodes are described as follows:

- Stage 1: The most significant 12 bits of the IP address are used to index the 12-bit segmentation table to obtain the absolute address of the root node of the corresponding multiway segment tree. Since the number of 144-bit nodes is less than 2^{17} for all the routing tables we experimented as shown in column 3 of Table 3, 17 bits are needed for representing the absolute address. The absolute address of the root node and the offset address of the DMST node to be accessed are summed up by an adder and output to the pipeline register of next stage. The address and data_in lines are used by the slow-path processor for update operations.
- Stage 2: The address of the DMST node to be accessed is used to read the off-chip SRAM memory to get the 144-bit data which will be stored in the stage 3 pipeline register and processed in the following stages.
- Stage 3: The real search process is performed in this stage which consists of a DMST node search logic shown in Fig. 8c. Four end points divide the node's address space into five intervals. Thus, the node search logic finds which interval matches the least significant 20 bits of the IP address. The node search logic returns a 5-bit search result in which only 1 bit corresponding to the matched interval is set.
- Stage 4: The O/N field selector uses the 5-bit search result computed in stage 3 to select the O/N field value of the matched interval. The lower part of the logic in this stage determines that if the DP processed in stage 3 is valid, it will replace the old default port found in the parent of the node currently processed. We assume the invalid default port is 0xFF. So, the implementation of this functionality is simply a combination of an 8-bit input AND gate and a 2×1 multiplexer.
- Stage 5: The finish flag is examined to determine whether the leaf node is reached in the search process. If the finish flag is off, the O/N field extracted from stage 4 is the child offset address of the current node which will be fed to the first stage pipeline register to start another run of the search process. Otherwise, when the finish flag is on, the search process of the current lookup operation is done. The final task is to extract the final next port number. If the next port number (i.e., the least significant 8 bits of the selected O/N field value) is not 0xFF, it will be the final port number implemented by the two concatenated NAND gates and a 2×1 multiplexer. Otherwise, the final next port will be the current default port computed in stage 4.

Finally, the design of the search engine for 10-way DMST is briefly described. Each node of the 10-way DMST is represented by a 288-bit format. A 12-bit segmentation table is also used to deal with the most significant 12 bits of all route prefixes. Further, as shown in the fourth column of Table 3, the number of 288-bit nodes of the largest DMST is less than 2^9. Thus, each O/N field only occupies 9 bits. In conclusion, the 288-bit format consists of nine 20-bit end points, ten 9-bit O/N fields, one 1-bit finish flag, one 8-bit default port field, and 9 bits are unused. The DMST node search logic shown in Fig. 8c in stage 3 is extended to compare 10 keys in parallel that can output a 10-bit search result.

5.1 Extending the DMST Search Engine

As stated above, each lookup operation of a packet will circulate through all the stages of the search engine n times if the destination IP address of the packet belongs to the address space of a multiway segment tree of n levels. Notice that n is at most six for the routing tables we experimented in this paper. Because the off-chip memory is cheap and the on-chip cost of the search engine is small, a straightforward extension is to avoid circulating through the same search engine many times by using n separate search engines connected serially. The resulting architecture will be a $5n$-stage pipelined search engine called *extended DMST search engine* that contains n separate off-chip SRAMs that can be accessed concurrently. Since $5n$ packets can be searched concurrently in the extended DMST search engine in one cycle, the throughput will be $1/r$, where r is the memory access delay of the off-chip SRAM.

6 Performance Evaluation

In this section, we present the experimental results for the proposed DMST IP lookup schemes. The simulation results in terms of memory requirement, search, and update times

TABLE 2
The Performance of All B-Tree-Based Schemes of Order 32

Routing Table		AS6447a	AS6447b	AS6447c	AS7660	AS2493
Year-month		2000-4	2002-4	2005-4	2005-4	2005-4
# of prefixes		79,560	124,824	163,574	159,816	157,118
DRAM (KByte)	MRT	2,640	4,117	5,368	5,232	5,150
	PIBT	2,568	4,006	5,223	5,091	5,012
	DMST	1,434	2,187	2,780	2,671	2,628
Search (μsec)	MRT	0.17	0.22	0.24	0.24	0.24
	PIBT	0.16	0.19	0.22	0.21	0.21
	DMST	0.15	0.18	0.20	0.20	0.20
Search (Mpps)	MRT	5.88	4.55	4.17	4.17	4.17
	PIBT	6.25	5.26	4.55	4.76	4.76
	DMST	6.67	5.56	5.00	5.00	5.00
Update (μsec)	MRT	1.03	1.12	1.18	1.17	1.17
	PIBT	0.83	0.91	0.96	0.96	0.96
	DMST	0.52	0.59	0.62	0.61	0.61

conducted on the general-purpose CPU, i.e., Intel Core 2 Quad Q9300 PC, are first given to show the performance of the proposed DMST implemented in pure software. Next, the performance of the proposed pipelined search engine for DMST based on Xilinx Virtex-5 XC5VLX330T FPGA is shown.

There are two parts of the operations that impact the update and search performance of the proposed scheme. The first part is the update operations of the dynamic multiway segment tree executed in the slow-path CPU presented in Section 4. The slow-path CPU runs in a full-fledged operating system that has its own memory (DRAM or SDRAM) different from the off-chip SRAMs accessed in stage 2 of the search engine. The multiway segment trees stored in the off-chip SRAM are duplicated in the slow-path memory. Each DMST node in the slow-path memory corresponds to a DMST node in the SRAM of the search engine. When a slow-path node is modified, it does not necessarily mean that the corresponding SRAM node also needs to be modified. For example, if a prefix is inserted in the *CSet* of a slow-path node and it is not the longest prefix in the *CSet*, the corresponding SRAM node remains intact. The second part of update operations is that the slow-path CPU computes which SRAM nodes need to be modified and perform the write operations on the off-chip SRAMs. We will explain the impact of updates on search performance is minimal later.

To measure the search time, the simulation IP traffic is obtained by first collecting the start addresses of all prefixes in the original routing table. Then, the IP addresses are randomized and fed them into our simulator that implements the proposed DMST scheme to obtain the average search times. In order to obtain the update performance results, the data structure for the proposed DMST scheme is first constructed according to the original routing table. Next, we randomly delete 10 percent prefixes from the structure we built and then insert these deleted 10 percent prefixes back into the structure. The average update times in the slow path and number of SRAM nodes to be modified are reported. The performance of the proposed DMST scheme is also compared with two existing B-tree-based schemes, namely, MRT [29] and PIBT [20]. The experimental results are based on five real-life IPv4 routing tables obtained from [4] and [27]. The simulation programs are written in C and gcc-3.2.2 compiler of Redhat 9.0 with an optimization level -O4 is used. The simulations are run on a 2.5-GHz Intel Core 2 Quad Q9300 PC that has 4×32 KB 8-way set associative L1 cache of 64-byte blocks, $2 \times 3{,}072$ KB 12-way set associative L2 cache of 64-byte blocks, and 4-GB main memory. The instruction called ReaD Time Stamp Counter (RDTSC) is also used to keep an accurate count of every clock cycle that occurs in the processor.

Table 2 and Fig. 9 present the performance results of order 32 for DMST, MRT [29], and PIBT [20]. The results for MRT and PIBT reported in [29] and [20] also use the order of 32. DMST consumes much less memory than MRT and PIBT because of the following reasons: First, DMST uses fewer keys and smaller nodes (without equal lists or heaps) than MRT and PIBT. Second, DMST and PIBT store each prefix at O(1) nodes per level (at most $O(logN)$ levels) and MRT stores each prefix at $\mathrm{O}(m)$ nodes per level, where m is the order of the B-trees. In terms of search, DMST is only 4-5 percent and 11-18 percent faster than MRT and PIBT, respectively. However, the update performance of DMST is 35-37 percent and 47-50 percent faster than MRT and PIBT, respectively. The average search performance is also shown in terms of Mpps. By using the minimal Ethernet packet size which is 64 bytes, the proposed DMST search algorithm with software implementation using Intel Core 2 Quad Q9300 PC can achieve the throughput of about 2.56-3.42 Gbps which is better than OC-48 (2.45 Gbps).

Table 3 shows the performance results of the proposed DMST search engine using 144-bit and 288-bit nodes based on Xilinx Virtex-5 XC5VLX330T FPGA containing 51,840 slices (each slice contains four LUTs and four flip-flops) and 324 *block RAM* blocks of 36 Kb each. The third and fourth columns show the total number of nodes and the number of nodes in the maximal segment for all the five routing tables. Column 5 summarizes the off-chip SRAM usage in Mb which is equivalent to the total number of

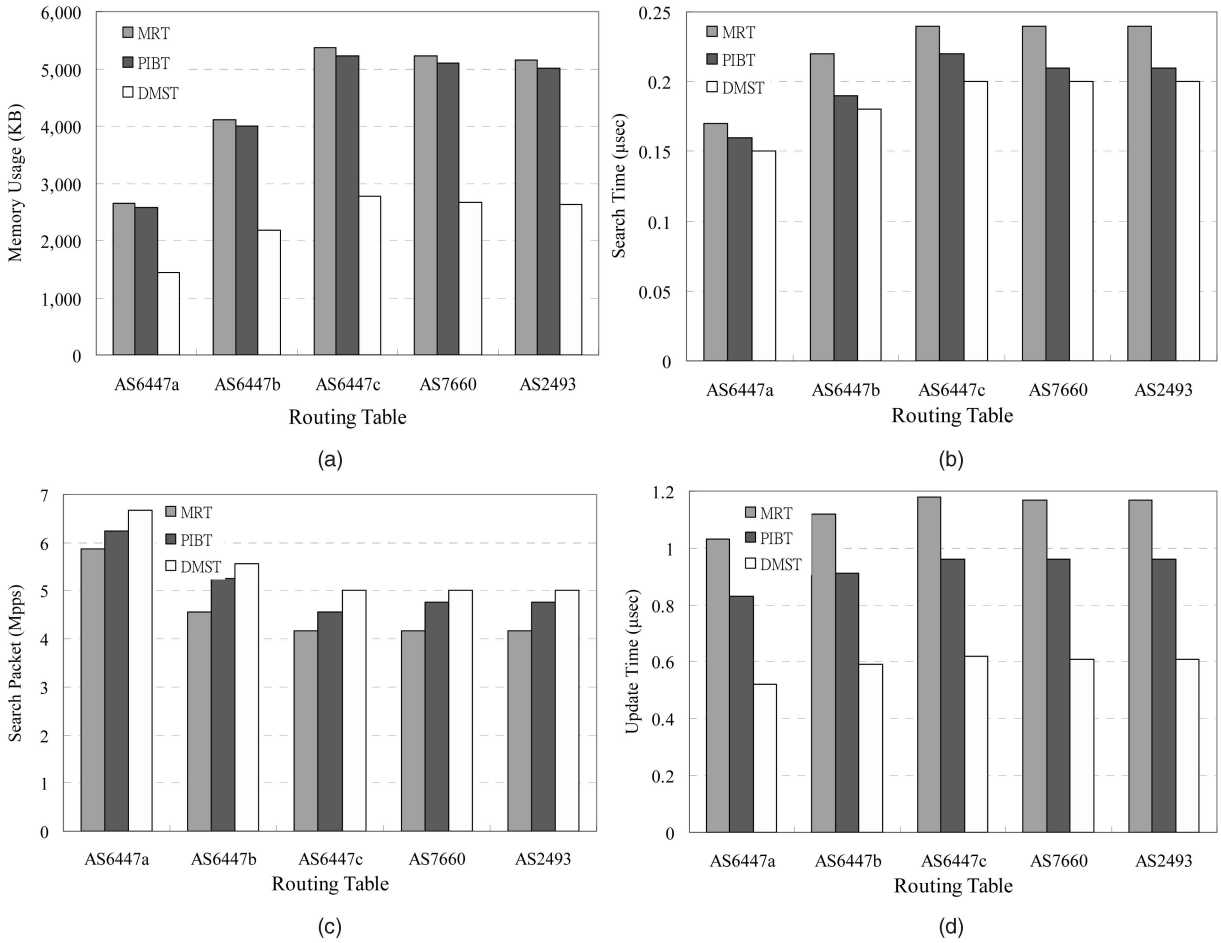

Fig. 9. Performance comparisons. (a) Memory usage in MByte. (b) Search time in μsec. (c) Search time in Mpps. (d) Update time in μsec.

TABLE 3
The Performance of DMST Search Engine, Where We Assume Packets are the Minimal Ethernet Packet of Size 64 Bytes

		Memory usage and on-chip cost				Search			Update
		number of nodes		off-chip SRAM (Mb)	on-chip configurable logic (slice)	pipeline cycle time (ns)	Number nodes accessed per search (avg/max)	throughput (average/worst-case)		average number of nodes modified per update
	Routing Tables	Total	Max. segment					Mpps	Gbps	
144-bit node (5-way)	AS6447a	45,400	788	6.23	463	5	4.50/6	44.4/33.3	22.8/17.1	2.75
	AS6447b	67,556	964	9.28			4.66/6	42.9/33.3	22.0/17.1	2.66
	AS6447c	85,018	1,036	11.68			4.68/6	42.7/33.3	21.9/17.1	2.52
	AS7660	81,662	874	11.21			4.57/6	43.8/33.3	22.4/17.1	2.51
	AS2493	80,532	870	11.06			4.59/6	43.6/33.3	22.3/17.1	2.52
288-bit node (10-way)	AS6447a	22,201	346	6.10	575	6	3.21/4	51.9/41.7	26.6/21.3	2.46
	AS6447b	31,880	430	8.76			3.29/4	50.7/41.7	25.9/21.3	2.44
	AS6447c	39,449	464	10.83			3.29/4	50.7/41.7	25.9/21.3	2.29
	AS7660	37,959	381	10.43			3.22/4	51.8/41.7	26.5/21.3	2.30
	AS2493	37,431	390	10.28			3.20/4	52.1/41.7	26.7/21.3	2.29
Note: the 12-bit segmentation table consumes 2 block RAM blocks										

nodes multiplied by the node size. For example, the search engine of 144-bit nodes for rule table AS6447a needs 6.23 Mb (i.e., $45{,}400 \times 144$ bits) of off-chip SRAM. We can see that the number of 144-bit nodes in the maximal segment tree of table AS6447c is 1,036 which is the largest among all the maximal segment trees of the routing tables we experimented. Thus, 11 bits are enough for O/N fields. Similarly, 9 bits in O/N field are enough for the search engine of 288-bit nodes. For the on-chip costs, column 6 shows that the search engines containing 144-bit and 288-bit nodes need

463 and 575 slices, respectively. As a result, the needed on-chip cost only accounts for one percentage of the on-chip configurable logic available in Xilinx Virtex-5 XC5VLX330T FPGA. Notice that the 12-bit segmentation table is implemented by block RAM which is faster than the implementation by the distributed RAM of the on-chip configurable logic. Based on our experiments, the 12-bit segmentation table consumes only two block RAM blocks.

For the search performance, we report the average numbers of nodes accessed per search and the worst case number of nodes accessed per search in the maximal segment tree in column 8 of Table 3. The latter represents the worst case number of off-chip memory accesses. Other than stage 2 that performs the read operations on off-chip SRAMs, stage 3 of the proposed pipelined architecture is the slowest stage which takes a processing delay of 4.94 and 5.11 ns for 144-bit and 288-bit nodes, respectively. Since the off-chip SRAMs only have a few operating frequencies available (e.g., 250, 200, 166, and 133 MHz), we have to select the one that meets the processing delay requirement of no smaller than 4.94 or 5.11 ns. Thus, the best operating frequencies of the off-chip SRAMs are 200 and 166 MHz for the pipelined architectures of 144-bit and 288-bit nodes, respectively. The processing time taken by stage 2 depends on I/O delays of both FPGA device and off-chip SRAM. In our case, Xilinx Virtex-5 XC5VLX330T FPGA has an output delay of 2.83 ns and an input delay of 2.03 ns. Cypress Pipelined Sync SRAM CY7C1347G-200Mhz [8] is a possible solution because of the following reasons: First, Cypress CY7C1347G has an input delay of 1.2 ns. By including the output delay of Xilinx Virtex-5 XC5VLX330T FPGA which is 2.83 ns, the total delay from FPGA to SRAM becomes 4.03 ns. Second, Cypress CY7C1347G has an output delay of 2.8 ns. By including the input delay of Xilinx Virtex-5 XC5VLX330T FPGA which is 2.03 ns, the total delay from SRAM to FPGA becomes 4.83 ns. Both the delays from FPGA to SRAM and from SRAM to FPGA are less than 5 ns. As a result, by using Cypress Pipelined Sync SRAM CY7C1347G-200Mhz as the choice of off-chip SRAM, the proposed pipelined architecture of 144-bit nodes can be set at the cycle time of 5 ns. However, for the pipelined architecture of 288-bit nodes, we have to use Cypress Pipelined Sync SRAM CY7C1347G-166Mhz and thus the cycle time of the proposed pipelined architecture is 6 ns.

With the cycle time bounded by 5 or 6 ns shown in column 7 of Table 3 and the number of nodes accessed per search shown in column 8, we can compute the throughput of the proposed pipelined search engine in terms of Mpps as shown in column 9. By assuming that all the packets have the minimal Ethernet packet size of 64 bytes, we can also calculate the throughputs in terms of billion bits per second (Gbps) in column 10. As a result, the search engines of 144-bit and 288-bit nodes can achieve the throughputs of 17.1 and 21.3 Gbps in the worst case, respectively. Finally, the average number of nodes that need to be modified per update is shown in the last column.

Subsequently, we shall describe that the impact of the update operations on the search performance is minimal. Refer to the stage 2 of pipelined search engine in Fig. 8a. The read operation on off-chip SRAM is performed in every cycle. We use the precedence model in [8] such that when a write operation is needed for an update along with a read operation in some cycle, it is given the precedence over the read operation. As a result, the read operation obtains the data written by the update instead of the correct data requested. Reading incorrect data from the off-chip SRAM only results in obtaining an incorrect next port for the corresponding packet which eventually will be dropped or forwarded to a wrong destination. Consider the rate of 1,000 updates per second for routing table AS6447a which needs 2.75 node modifications per update as shown in the last column of Table 3. These node modifications will cause at most 2,750 packets being dropped or forwarded incorrectly. Compared to the 44.4 million packets being forwarded by the proposed search engine containing 144-bit nodes for table AS6447a, 2,750 (the number of packets being dropped or forwarded incorrectly) is very small and can be ignored. Thus, the throughput of the router remains almost the same when updates are taken into consideration.

The performance results of the extended DMST search engine are shown in Table 4. The pipeline cycle time is bounded by the stage 2 which is 5 and 6 ns for the search engines of 144-bit and 288-bit nodes, respectively. When all the packets are the minimal Ethernet packets of size 64 bytes, the overall throughputs will be 102 and 85 Gbps for the search engines of 144-bit and 288-bit nodes, respectively. Note that the throughputs are independent of routing table sizes. In Table 4, we also show the total number of nodes needed in each level. For example, there are 4,096 and 2,997 144-bit nodes in levels 1 and 2 of the five-way search engine, respectively, for routing table AS6447a. The update speed will remain the same as before.

In practice, the idea of using multiple off-chip SRAMs in the extended DMST search engine to speed up the search performance is limited by the number of I/O pins provided FPGA devices. If a single FPGA device provides less number of I/O pins than needed, we have to use multiple FPGA devices. In our case, Xilinx Virtex-5 XC5VLX330T FPGA has only 960 I/O pins which is less than needed (about 1,087 pins) for the extended architecture of 144-bit nodes with four off-chip SRAMs. A simple solution is to use two Xilinx Virtex-5 XC5VLX330T FPGA devices. Since there is one more stage needed between the two FPGA devices, the total number of stages in the extended DMST search engine is 31 instead of 30.

7 Summary and Future Work

We developed a DMST-based data structure for IP lookups. DMST is implemented with a B-tree for dynamic forwarding tables. DMST has the same asymptotic complexities as MRT and PIBT. With the experiments using real IPv4 routing tables, DMST is found to be better than MRT and PIBT in terms of search speed, update speed, and memory consumption because DMST uses smaller nodes and fewer keys. We also proposed a pipelined search engine using off-chip SRAMs to further improve the search performance. By utilizing the current FPGA and off-chip SRAM technologies, the proposed five-stage pipelined search engine can achieve the worst case throughputs of 33.3 and 41.7 Gbps for the search engines of 144-bit and 288-bit nodes, respectively. These throughputs are equivalent to 17.1 and 21.3 Gbps for

TABLE 4
The Performance of Extended DMST Search Engine, Where We Assume Packets are the Minimal Ethernet Packets of size 64 Bytes

	Routing Tables	Number of nodes							off-chip SRAM (Mb)	on-chip configurable logic (slice)	on-chip block RAM (block)	Search Throughput	
		Total	Level #									Mpps	Gbps
			1	2	3	4	5	6					
144-bit node (5-way)	AS6447a	45,400	4,096	2,997	5,227	12,404	19,785	891	6.23	2,778	12	200	102
	AS6447b	67,556	4,096	3,431	7,883	17,326	30,934	3,886	9.28				
	AS6447c	85,018	4,096	3,945	10,042	22,084	40,099	4,752	11.68				
	AS7660	81,662	4,096	4,173	10,833	22,776	37,910	1,874	11.21				
	AS2493	80,532	4,096	4,170	10,590	22,032	36,689	2,955	11.06				
288-bit node (10-way)	AS6447a	22,201	4,096	3,084	9,452	5,569	N/A		6.10	2,300	8	167	85
	AS6447b	31,880	4,096	4,157	13,850	9,777			8.76				
	AS6447c	39,449	4,096	5,183	18,165	12,005			10.83				
	AS7660	37,959	4,096	5,530	18,162	10,171			10.43				
	AS2493	37,431	4,096	5,571	18,328	9,436			10.28				

minimal Ethernet packets of size 64 bytes. Furthermore, the extended DMST search engine of 144-bit and 288-bit nodes can achieve the throughputs of 200 and 167 Mpps, respectively. Similarly, these throughputs are equivalent to 102 and 85 Gbps for minimal Ethernet packets of size 64 bytes.

In summary, the proposed pipeline architecture consists of two major advantages. First, the stage 2 that performs off-chip memory accesses can be incorporated with any advanced memory technologies, and thus a very high-throughput router can be achieved. Second, the proposed architecture is based on the data structure that can be updated and thus it also has a very good update performance.

Although the current proposal provides a very high-throughput router design along with its updatable feature, some enhancements can be done in the future. First, since IPv6 addresses are 128 bits, the order of the B-tree constructed for IPv6 addresses will become much smaller than that for IPv4 addresses. This leads to a deeper B-tree structure and thus reduces the search speed. Therefore, what remains to be explored is to apply the insights from our IPv4-based pipeline architecture to incorporate IPv6 specification with the current design. Second, we can construct two and four parallel pipelines by using dual-port and quad-port memories, respectively. This enhancement provides a straightforward mechanism to double or quadruple the throughput of the current design. Third, other than stage 2 for off-chip memory accesses, the bottleneck stage of the proposed pipeline architecture is stage 3 in which DMST node search logic takes most of the time needed (i.e., 4.94 and 5.11 ns obtained on Xilinx Virtex-5 XC5VLX330T FPGA for 144-bit and 288-bit nodes, respectively). Therefore, an optimization technique is needed to reduce the delay of this stage. Splitting stage 3 into two substages may be a possible solution. This optimization for stage 3 becomes even more important as the memory access speed at stage 2 will increase in the future. Fourth, the update performance can also be improved by using the embedded processors implemented on FPGA which can reduce the communication overhead over the I/O interface between the FPGA and external processor in the current design shown in Fig. 2. For example, Xilinx Virtex-5 FXT FPGAs contain 1 to 2 PowerPC 440 Microprocessors. Thus, the functionalities of the slow and fast paths can all be built into one single Xilinx Virtex-5 FXT FPGA chip. Fifth, the power consumption is also an issue that needs to be explored, especially when multiple off-chip SRAMs are used. Comparing the power consumption of off-chip memory over on-chip memory may be a possible direction.

ACKNOWLEDGMENTS

The authors would like to express their sincere thanks to the editors and the reviewers, who gave very insightful and encouraging comments. This work was supported in part by the National Science Council, Republic of China, under Grant NSC-96-2221-E-006-190-MY3.

REFERENCES

[1] Avnet Electronics Marketing—Electronic Components Distributor Services, http://avnetexpress.avnet.com/, 2009.
[2] F. Baker, "Requirements for IP Version 4 Routers" RFC 1812, June 1995.
[3] M.D. Berg, M.V. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications,* second ed. Springer Verlag, 2000.
[4] BGP Routing Table Analysis Reports, http://bgp.potaroo.net/, 2009.
[5] 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.
[6] H. Chao, "Next Generation Routers," *Proc. IEEE,* vol. 90, no. 9, pp. 1518-1558, Sept. 2002.
[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms,* second ed. The MIT Press, Sept. 2001.
[8] Cypress Semiconductor Corp., "CY7C1347G, 4-Mbit (128K x 36) Pipelined Sync SRAM," Document #: 38-05516 Rev. *F, Revised, Jan. 2009.
[9] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM SIGCOMM,* pp. 3-14, Oct. 1997.
[10] DigiKey Corp.—Electronic Components Distributor, http://www.digikey.com/, 2009.

[11] A. Feldman and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. IEEE INFOCOM,* vol. 3, pp. 1193-1202, Mar. 2000.
[12] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. IEEE INFOCOM,* vol. 3, pp. 1240-1247, Apr. 1998.
[13] P. Gupta, B. Prabhakar, and S. Boyd, "Near-Optimal Routing Lookups with Bounded Worst Case Performance," *Proc. IEEE INFOCOM,* vol. 3, pp. 1184-1192, Mar. 2000.
[14] I. Ioannidis, A. Grama, and M. Atallah, "Adaptive Data Structures for IP Lookups," *Proc. IEEE INFOCOM,* vol. 1, pp. 75-84, 2003.
[15] C. Labovitz, G. Malan, and F. Jahabnian, "Internet Routing Instability," *Proc. ACM SIGCOMM,* vol. 6, no. 5, pp. 515-528, Oct. 1998.
[16] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Trans. Networking,* vol. 7, no. 3, pp. 324-334, June 1999.
[17] 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.
[18] H. Lu and S. Sahni, "$O(\log n)$ Dynamic Router-Tables for Prefixes and Ranges," *IEEE Trans. Computers,* vol. 53, no. 10, pp. 1217-1230, Oct. 2004.
[19] H. Lu and S. Sahni, "Enhanced Interval Trees for Dynamic IP Router-Tables," *IEEE Trans. Computers,* vol. 53, no. 12, pp. 1615-1628, Dec. 2004.
[20] H. Lu and S. Sahni, "A B-Tree Dynamic Router-Table Design," *IEEE Trans. Computers,* vol. 54, no. 7, pp. 813-824, July 2005.
[21] E. McCreight, "Priority Search Trees," *SIAM J. on Computing,* vol. 14, no. 2, pp. 257-276, 1985.
[22] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE J. on Selected Areas in Comm.,* vol. 17, no. 6, pp. 1083-1092, June 1999.
[23] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network,* vol. 15, no. 2, pp. 8-23, Mar./Apr. 2001.
[24] S. Sahni and K. Kim, "An $O(\log n)$ Dynamic Router-Table Design," *IEEE Trans. Computers,* vol. 53, no. 3, pp. 351-363, Mar. 2004.
[25] R. Sangireddy, N. Futamura, S. Aluru, and A.K. Somani, "Scalable, Memory Efficient, High-Speed IP Lookup Algorithms," *IEEE/ACM Trans. Networking,* vol. 13, no. 4, pp. 802-812, Aug. 2005.
[26] 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.
[27] University of Oregon Route Views Archive Project, http://archive.routeviews.org/, 2009.
[28] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *Proc. ACM SIGCOMM,* pp. 25-36, Oct. 1997.
[29] P. Warkhede, S. Suri, and G. Varghese, "Multiway Range Trees: Scalable IP Lookup with Fast Updates," *Computer Networks,* vol. 44, no. 3, pp. 289-303, Feb. 2004.
[30] Xilinx, "Virtex-5 Family Overview," Product Specification, DS100 (v5.0), Feb. 2009.

Yeim-Kuan Chang received the MS degree in computer science from the University of Houston at Clear Lake in 1990, and the PhD degree in computer science from Texas A&M University, College Station, in 1995. He is currently an associate professor in the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan. His research interests include Internet router design, computer networking, computer architecture, and multiprocessor systems. He is a member of the IEEE Computer Society.

Yung-Chieh Lin received the MS degree in computer science and information engineering from the National Cheng Kung University, Taiwan, Republic of China, in 2005. He is currently working toward the PhD degree in computer science and information engineering at the National Cheng Kung University, Taiwan, Republic of China. His current research interests include high-speed networks and high-performance Internet router design.

Cheng-Chien Su received the MS degree in computer science and information engineering from the National Cheng Kung University, Taiwan, Republic of China, in 2005. He is currently working toward the PhD degree in computer science and information engineering at the National Cheng Kung University, Taiwan, Republic of China. His research interests include high-speed packet processing in hardware and deep packet inspection architectures.