IEEE WORKSHOP ON
HIGH-PERFORMANCE SWITCHING AND ROUTING

Novel Hardware Architecture for Fast Address Lookups

Pronita Mehrotra and Paul D. Franzon

ABSTRACT

For every packet an IP router receives, it makes a routing decision based on the packet's destination address. The router's forwarding rate is usually limited by the rate at which it can make these decisions. We describe a new method for implementing route lookups in hardware. Our method can be implemented in the forwarding engine of a network processor or router using a small on-chip SRAM and an off-chip DRAM, and it achieves a rate of one lookup per DRAM random access time. We present our method and discuss an implementation that uses a DRAM with 64 ns random access time to give over 15 million lookups per second. Our tests show that the method performs well for realistic routing tables while using only modest amounts of memory.

INTRODUCTION

When an IP router receives a packet on one of its input ports, it must decide to which output port the packet will be forwarded. To make this decision, it has to match the packet's destination address against a database of destination networks and hosts. This database is generated by the router from a routing table, each entry of which consists of a variable-length prefix and a corresponding next hop address. In order to forward a packet, the router needs to locate the next hop address corresponding to the longest matching prefix in the routing table for the packet's destination address, and send the packet to the appropriate output port.

For instance, a routing table can have an entry of 128.* with an associated output port of 2, and another entry of 128.14.* with an output port of 5. The rules are interpreted as follows: if the destination address begins with 128 the packet should be sent to output port 2, unless the destination address begins with 128.14, in which case the packet should go to output port 5.

The work presented in this article describes a new method for performing route lookups quickly and efficiently. It consists of two parts: an algorithm for generating a routing database from a given routing table, and an algorithm for searching this database. Two main factors motivated this work. First, with current technology, terabit routing requires lookups to be performed in hardware. Second, a constant lookup time is especially important for emerging applications like optical burst switching (OBS) that rely on good estimates of time taken from source to destination.

As routing databases grow larger, more memory accesses are required to determine the next-hop address for a given destination. Moreover, a large routing database cannot be stored in an on-chip memory or a cache; therefore, very expensive off-chip accesses are required. Our aim was to design a route lookup method that minimizes the off-chip accesses while maintaining a fast constant lookup time. Our search algorithm requires several accesses to a small fast on-chip SRAM and only one access to a slower DRAM in order to determine the next-hop address.

In the rest of this article, we look at some existing approaches to the route lookup problem. We then describe our algorithms for generating and searching the routing database. We also discuss a possible hardware implementation, and some performance and design issues.

RELATED WORK

A number of approaches have been used to search for longest matching prefixes. Most approaches fall into one of two classes [1], both of which use tree-like structures to store the routing database. In *search trie* methods, each bit in the address is checked, with a 0 bit pointing to the left subtree and a 1 pointing to the right subtree. In *search tree* methods, the destination address is compared to the median value of each subtree. If the address is less than the median value, it is directed to the left subtree; if larger, it is pointed to the right subtree.

Existing trie-based schemes include direct and indirect lookups [2]. Both of these require large amounts of memory to store the forward-

0163-6804/02/$17.00 © 2002 IEEE

ing tables. The number of lookups is small (1–2), but these schemes do not scale well with size. Binary tries, on the other hand, store data fairly efficiently. However, they require a large number of memory accesses compared to the direct or indirect lookup schemes. Variations of the basic binary trie like Patricia [3] and LC tries [4] improve performance to some extent, but the average number of memory accesses is still fairly large. Techniques that use content addressable memories (CAMs) are not suitable for large routing databases.

Other suggested approaches include variations of binary search. The length of the search in these approaches depends on the number of entries in the routing table, as in [5]. Waldvogel *et al.* [6] suggest a hash-based scheme where a binary search is performed over possible prefix lengths.

Description of the Algorithm

Our scheme compacts the trie data structure so that it is small enough to fit on an on-chip SRAM. A final off-chip DRAM access is required to read the next-hop address.

Data Structure

We build the SRAM and DRAM databases from the conventional multiway trie structure. The SRAM database contains information which represents the topology of the trie, while the DRAM contains the next-hop addresses corresponding to the leaves of the trie.

In addition to the SRAM and DRAM databases, we also maintain an array (*Level*) in SRAM. The *i*th entry of *Level* points to the bit in the SRAM database where the *i*th trie level starts.

The route lookup is done in two stages. In the first stage the SRAM is used to traverse to the longest matching leaf node in the trie, while in the second stage the DRAM is read to get the next-hop address.

Building the Data Structure

The data structure to be stored in the SRAM and DRAM are built from the corresponding multiway trie. We describe an implementation using a 16-way trie, although any degree of trie can be built. The trie is built as follows:

Step 1: Read each entry from the routing table and store it in a list. Sort the list in ascending order. For prefixes of different lengths where one prefix forms the beginning of the other, the prefix with the smaller length is considered to be smaller. For example, 10* is considered smaller than 100*. This ensures that while building the trie, parent nodes are processed before child nodes.

Step 2: Create the root node of the trie and initialize the child node pointers to NULL.

Step 3: Read each entry from the list and expand if necessary to complete the trie (to make sure that every internal node has X children, where X is the trie degree). Add appropriate nodes to the trie along with their next-hop addresses.

Step 4: Once the trie is built, construct the SRAM and DRAM data structures and the array *Level* by doing a breadth-first traversal on the trie. The SRAM is built by writing a 1 for every internal node and a 0 for leaf nodes, as shown in Fig. 1. While constructing the SRAM data, the first 1, which represents the root node, is assumed; there is no need to store it. The DRAM is built by writing an entry for every node in the trie in a breadth-first order.

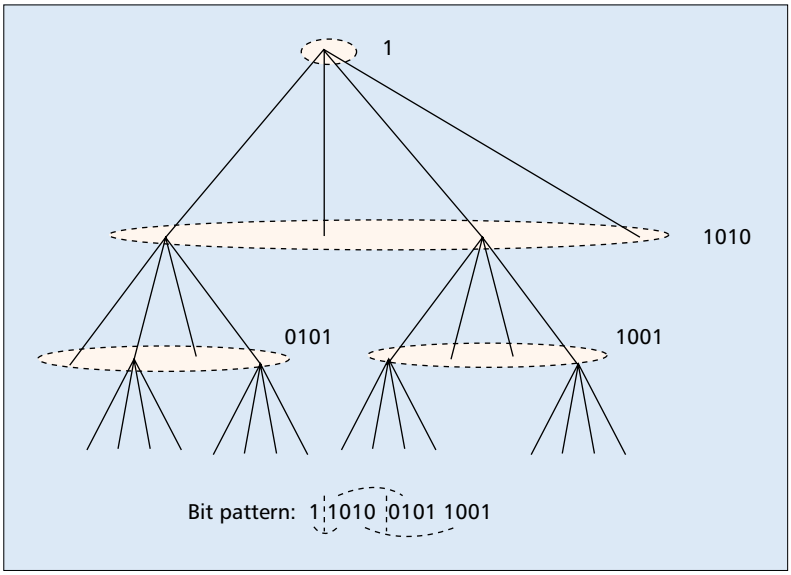

Figure 1. *A sample four-way trie and the corresponding bit pattern.*

The depth of the trie is (*No. of Address bits*)/($\log_2 X$), where X is the degree of the trie. This is also the number of lookups required during insertion of an entry into the trie in the worst case. Since building the trie requires inserting N entries, where N is the total number of entries in the routing table, the total number of memory lookups while building the trie is $N * D$, where D is the depth of the trie.

Searching the Data Structure

The algorithm to search for the longest matching prefix for a given address is summarized below. The degree of the trie is denoted by X. *Level*[i] contains the SRAM bit where the *i*th level of the trie begins.

Step 1: Initialize the start pointer (*START*) to point to the beginning of the SRAM database and set *PREV* and i to 0.

Step 2: Starting at bit $i * \log_2(X)$, read $\log_2(X)$ bits of the destination address and assign this value to *OFFSET*.

Step 3: Read the bit at SRAM position *START* + *OFFSET*.

Step 4a: If the bit read is 1, count the number of 1s between SRAM locations *Level*[i] and (*START* + *OFFSET* – 1), inclusive, and assign this to *ONES*. Set *PREV* to the current value of (*START* + *OFFSET*). Move the *START* to *Level*[i + 1] + (*ONES* $*$ X). Increment i and repeat steps 2–4.

Step 4b: If the bit read is 0, the search terminates. The index of the next-hop address in the DRAM database is given by ($P1 * X$) + *OFFSET*, where $P1$ is the number of 1s between SRAM locations 0 and PREV, inclusive.

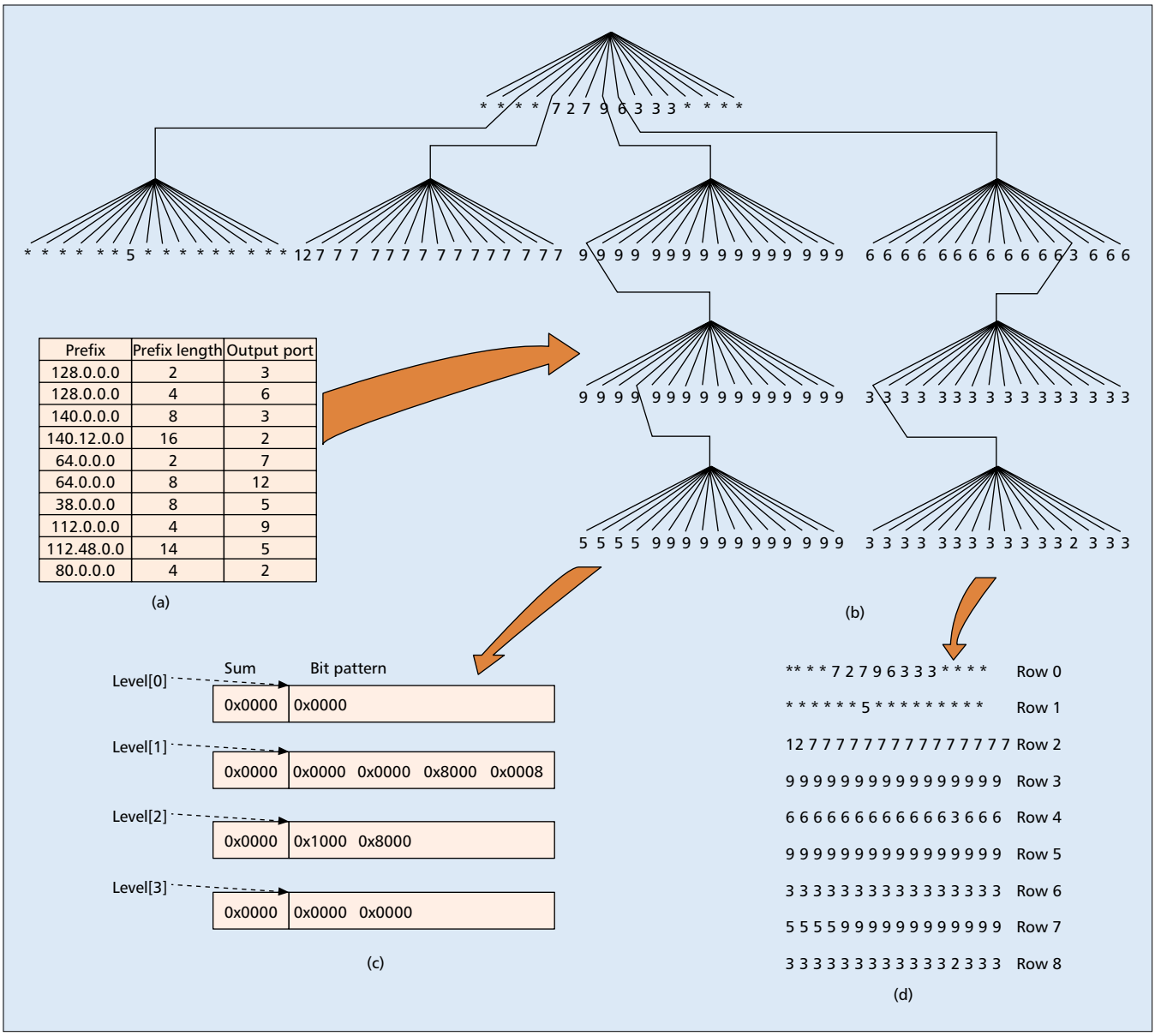

Figure 2. *An example of a 16-way trie: a) sample database of prefixes and associated hops; b) 16-way trie for the prefixes; c) bit pattern as stored in the SRAM; d) data stored in the DRAM.*

Example of a Search on a 16-Way Trie — Figure 2 shows how the SRAM and DRAM data is constructed for a 32-bit routing table using a 16-way trie. Figure 2a shows a sample set of prefixes along with their prefix lengths and the output ports. The 16-way trie for the sample set of prefixes is shown in Fig. 2b, where the numbers below the node indicate the output ports. Memory cells in SRAMs and DRAMs are arranged in rows and columns. An entire row of an SRAM (128 bits in our implementation) can be read with a single access, whereas only a single column can be read at a time from a DRAM. The bit pattern the SRAM stores for each of the levels is shown in Fig. 2c along with the array Level. The off-chip DRAM stores only the next-hop addresses as shown in Fig. 2d, where the * represents the default next hop. Each DRAM row contains 16 columns containing 16 output ports.

For the example in Fig. 2 child nodes 2,4,7, and 8 of the root node are the only internal nodes, and this is represented in the SRAM as binary "0010 1001 1000 0000" or hexadecimal 0x2980. Similarly, in the next level, these nodes while going from left to right give rise to the SRAM bit pattern 0x0000 0x0000 0x8000 0x0008. The data stored in the DRAM maps directly onto the trie in the same breadth-first order.

We introduce two additional data structure to speed up our implementation. In our SRAM database, we set aside the first few bits of each SRAM row for an extra "Sum" field. This field contains the number of 1 bits that are on the current trie level and appear in previous SRAM rows. This way, even if a trie level spans multiple SRAM rows, we never need more than one SRAM access to read an SRAM bit and compute the number of 1s before it on its trie level (i.e., for steps 3 and 4a in the search algorithm). Second, we store the total number of 1s for each level in a separate array. This helps in computing the final DRAM offset (step 4b in the search algorithm).

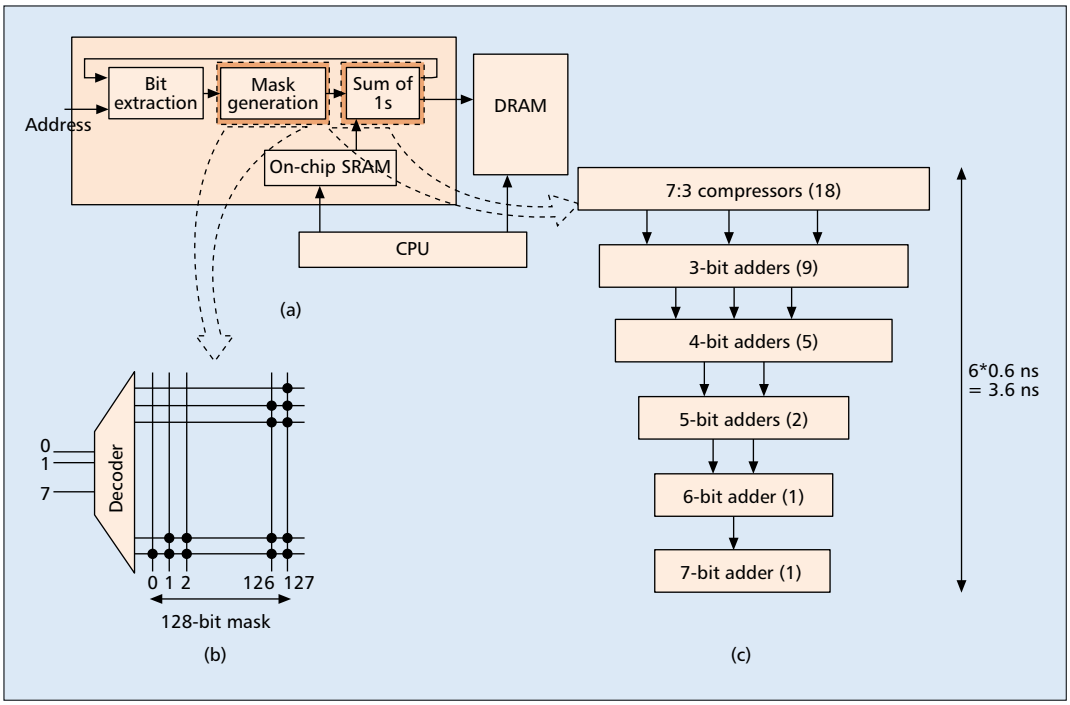

Figure 3. *Hardware implementation of the forwarding engine: a) a block diagram of the forwarding engine; b) generation of a mask from the bit position; c) adders used in the computation of the sum of 1s.*

We build the SRAM and the DRAM databases from the conventional multiway trie structure. The SRAM database contains information which represents the topology of the trie, while the DRAM contains the next-hop addresses corresponding to the leaves of the trie.

Now suppose that a packet arrives with a destination IP address of 112.48.32.248 (i.e., 0x703020f8). The longest prefix search proceeds as follows, where the step numbers correspond to those given in the search algorithm.

(Step 1) We initialize by setting $START = 0$, $i = 0$, $PREV = 0$, $X = 16$.

(Step 2) The search process starts by looking at the first 4 bits of the address (0111), which gives an *OFFSET* of 7.

(Step 3) Since $START + OFFSET = 7$, we look at bit 7 in the bit pattern for level 0. As a "1" is stored there, the search continues.

(Step 4a) There are two 1s in the bit pattern before bit 7 in level 0. This means that for the next level, (level 1), the first $2 * 16$ bits have to be skipped. Therefore, $ONES = 2$, $PREV = 7$, $START = Level[1] + (2 * 16)$, $i = 1$.

(Step 2) The next 4 bits of the address (0000) give an $OFFSET = 0$.

(Step 3) This leads to the starting bit of the pattern 0x8000 in level 1. Since a "1" is stored there, the search is not complete.

(Step 4a) Since there are no 1s before bit 0 in level 1, no bits need be skipped for the next level. The corresponding variables take the values $ONES = 0$, $PREV = Level[1] + 32$, $START = Level[2] + (0 * 16)$, $i = 2$.

(Step 2) Since the next four bits of the address are 0011, *OFFSET* gets a value of 3.

(Step 3) This points to bit 3 in the pattern 0x1000, which is set to 1.

(Step 4a) As before, there are no 1s before bit 3 in level 2; therefore, no bits are skipped for the next level. We set $ONES = 0$, $PREV = Level[2] + 0$, $START = \text{Level}[3] + (0 * X)$, $i = 3$.

(Step 2) The next four bits of the address are 0000 which gives an *OFFSET* of 0.

(Step 3) The SRAM bit pointed to is now bit 0 in the bit pattern 0x0000 of level 3. Now this bit is set to "0," which means that the search process terminates.

(Step 4b) As there are a total of 7 1s (starting from beginning up to the bit pointed to by $Level[2] + 0$) in the SRAM database, the 112th (i.e., $\mathbf{P}1 * X + OFFSET$) DRAM entry needs to be looked up. In this example, the DRAM row address is also the total number of 1s and the column address is the offset, which means that the entry stored in row 7 and column 0 of the DRAM needs to be read. A 5 is stored there, and that is the output port corresponding to the longest matching prefix.

Hardware Implementation

Modern router designs achieve higher message handling capacity by placing more intelligence in the line cards. Each line card contains a copy of the forwarding table, which is generated from the main routing table and updated periodically. Operations like address lookup, scheduling and configuring the switch fabric are performed by the line cards themselves.

The block diagram of the forwarding engine and the details of various sub-blocks for a 16-way implementation are shown in Fig. 3. The bit extractor picks the next 4 bits of the address. This is used to generate a mask for computing the sum of 1s. The sum of 1s unit takes the mask and the SRAM row to determine the next offset. Once the SRAM traversal is complete, a read request for the off-chip DRAM is generated, and after the DRAM access time, the next-hop address is available. The data stored in the SRAM and DRAM is generated in software.

The SRAM traversal is implemented as a finite state machine (FSM) with two states. In

The SRAM traversal is implemented as a Finite State Machine with two states. In the first state an entire SRAM row is read and in the second state the mask is generated and the sum of 1s computed.

Site	No. of entries	SRAM (kbytes)	DRAM (Mbytes)	Trie Memory (Mbytes)	Bytes/entry
MaeEast	23,113	24.4	11.43	24.28	1.08
MaeWest	35,752	34.75	16.32	34.683	1.99
PacBell	27,491	29.08	13.66	29.03	1.08
Paix	17,641	20.5	9.63	20.46	1.19
AADS	31,958	32.25	15.15	32.18	1.03

Table 1. *Memory requirements for various routing tables.*

the first state an entire SRAM row is read and in the second state the mask is generated and the sum of 1s computed. Each of the two states takes 8 ns to complete, and a total of 16 ns is taken to traverse one level in the SRAM bit pattern. Since there are a total of 8 levels to be traversed in the SRAM (for 32-bit addresses in IPv4), it takes 128 ns to traverse the SRAM. The FSM loop can be unrolled and pipelined more than once to increase the throughput. In our implementation, we unrolled the FSM loop once and pipelined the two FSMs, to give results every 64 ns. This was done to match the random DDR DRAM access time of 64 ns.

Generating the Mask

To compute the sum of 1s till a certain bit position, we generate a mask to remove the unwanted bits from the SRAM row. The result obtained after bitwise ANDing the mask with the SRAM row is given as the input to the unit computing the sum of 1s.

To generate the mask, the bit position is first decoded and, depending on the 8-bit input, one of the output bits of the decoder goes high. The output of the decoder feeds into the mask generator circuit, as shown in Fig. 3b. The delay through the generator is the maximum delay at line 127 with a fanout of 128. The 8:128 bit decoder takes around 0.7 ns (in 0.18 µ technology) to decode, while the mask generator again takes around 0.6 ns to complete.

Computing the Sum of 1s

Sum of 1s can be computed in a number of ways. The simplest way is to use a bank of adders. For a 128-bit-wide SRAM row, required adders are shown in Fig. 3c. The 7:3 compressors used in the first row add up 7 1-bit numbers and reduce the result to a 3-bit number. For 0.18 µ technology, a 32-bit adder takes about 0.6 ns. We have kept the same budget for our smaller adders, even though smaller adders take less time. The total worst-case time taken to compute the sum is less than 4 ns. In all, the total time taken to compute the sum of 1s is well within the budget time of 8 ns for each FSM state.

Performance of the Scheme

We ran the algorithm on practical routing tables from [7]. The results have been summarized in Table 1, which shows the amount of memory required for these routing tables. For instance, the MaeEast routing table with over 23,000 entries takes around 25 kbytes of SRAM to store the bit pattern and around 12 Mbytes of DRAM to store the next-hop addresses. In a conventional trie implementation, around 25 Mbytes of DRAM memory (the second last column in the table) would be required. The last column in the table shows the amount of compaction that can be achieved in the on-chip SRAM. For all the routing tables around 1 byte of SRAM memory per entry in the routing table is required. This gives very good scalability, which will be very important when routing tables become even larger in the future.

Discussion

The required SRAM is small enough (about 35 kbytes for a routing database > 30,000 entries) to easily fit on a chip. The data in our case is compacted to around 1 byte for every entry in the routing table. Also, the overall memory consumption (SRAM and DRAM) using this scheme is almost half that required in conventional implementations. The total CPU time taken to build the SRAM and DRAM data is on the order of 100 ms on a Sun Ultra 5 with a 333 MHz processor. Updating a route only requires changing an entry in the DRAM. Adding or deleting a prefix, on the other hand, requires the

Site	No. of entries	Degree = 16 KB (bytes/entry)	Degree = 8 KB (bytes/entry)	Degree = 4 KB (bytes/entry)	Degree = 2 KB (bytes/entry)
MaeEast	23,113	24.4(1.08)	16(0.71)	8.09(0.36)	6.57(0.29)
MaeWest	35,752	34.75(1.99)	23.03(0.66)	11.41(0.33)	9.23(0.26)
PacBell	27,491	29.08(1.08)	19.24(0.72)	9.76(0.36)	7.99(0.3)
Paix	17,641	20.5(1.19)	13.09(0.76)	6.86(0.4)	5.6(0.33)
AADS	31,958	32.25(1.03)	21.33(0.68)	10.67(0.34)	8.67(0.28)

Table 2. *SRAM Requirements for different degrees of the Trie structure.*

data structure to be built from scratch. Since most forwarding tables need to be updated only about once every second, building the entire database from scratch is not an issue.

Each route lookup in our implementation requires 8 SRAM accesses and 1 DRAM access. The number of SRAM accesses can be reduced further by splitting the SRAM and performing a direct lookup on the first 16 bits. This would reduce the number of SRAM accesses to 5. These are pipelined so that the DRAM cycle time is the limiting factor. By implementing queues and multiple DRAMs in parallel, an even higher throughput can be obtained. In our current implementation with a single DRAM, a lookup can be done every 64 ns, which gives over 15 million lookups/s. In a conventional implementation, 8 DRAM accesses would be required.

The overall performance of the forwarding engine in terms of throughput is constant for different degrees of the trie. The amount of SRAM required for various degrees of the trie is shown in Table 2. Efficiency of memory compaction increases with decreasing degree of the trie (fewer SRAM bytes per entry are required) due to less wastage during trie completion. The total latency of the address lookup increases with a smaller degree trie as more SRAM accesses are required to traverse an increased number of trie levels. A more deeply pipelined FSM would be required to maintain the same throughput. Our implementation chose a 16-way trie in order to reduce the latency and keep the hardware simple. This comes at a cost of higher memory consumption.

A wider SRAM, like 512 or 1024 bits wide, could be used in the design without affecting the throughput. This reduces the memory overhead of the forwarding engine at the expense of additional hardware. In the current implementation, 20 bits are used to hold the sum of 1s value for every 128 bits of data in the SRAM row. The memory overhead in the design is an additional 15–16 percent. By going to a design using 512-bit-wide SRAM, the memory overhead can be reduced to under 4 percent.

Conclusions

In this article we present a fast efficient address lookup scheme that is easy to implement in hardware. The scheme involves using a small on-chip SRAM to store a compacted version of the table, and an off-chip DRAM to store the next-hop address. The throughput of the scheme is limited solely by the single DRAM memory cycle required per access. The amount of SRAM required is quite small, and on practical routing tables, 1 byte of SRAM memory is required per entry in the table (for a 16-way trie). The operation of the SRAM and DRAM is pipelined such that a lookup can be done every 64 ns, giving a lookup rate of over 15 million/s.

A wider SRAM like 512 or 1024 bits wide, could be used in the design without affecting the throughput. This reduces the memory overhead of the forwarding engine at the expense of additional hardware.

Acknowledgments

The authors would like to acknowledge the following funding sources: ARDA under contract MDA904-00-C-2133 and NSF under contract EIA-9703090.

References

[1] N. McKeown and B. Prabhakar, "High Performance Switches and Routers: Theory and Practice," *Hot Interconnects Tutorial Slides*; http://tiny-tera.stanford.edu/nickm/talks/index.html), Aug. 1999.
[2] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. IEEE INFOCOM '98*, San Francisco, CA, 1998, pp. 1382–91.
[3] K. Sklower, "A Tree-Based Routing Table for Berkeley Unix," Tech. rep., UC Berkeley.
[4] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE JSAC*, vol. 17, June 1999, pp. 1083–92.
[5] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups using Multiway and Multicolumn Search," *Proc. IEEE INFOCOM '98*, vol. 3, San Francisco, CA, 1998, pp. 1248–56.
[6] M. Waldvogel *et al.*, "Scalable High Speed IP Routing Lookups," *Proc. ACM SIGCOMM*, vol. 27, Oct. 1997, pp. 25–36.
[7] "Michigan University and Merit Network. Internet Performance Management and Analysis (IPMA) Project," http://nic.merit.edu/ipma.

Biographies

Pronita Mehrotra (pmehrot@anr.mcnc.org) received her Master's degree in electrical engineering from the Indian Institute of Technology, Bombay, in 1997 and her Ph.D. degree from North Carolina State University in 2002. She is currently working as a hardware researcher in the Advanced Networking Research group at MCNC, North Carolina, where her focus is on optical burst switched networks.

Paul D. Franzon (paulf@ncsu.edu) is currently a professor in the Department of Electrical and Computer Engineering at North Carolina State University. He has over 10 years experience in electronic systems design and design methodology research and development. During that time, in addition to his current position, he has worked at AT&T Bell Laboratories, Holmdel, New Jersey, at the Australian Defense Science and Technology Organization, as a founding member of a successful Australian technology startup company, and as a consultant to industry, including technical advisory board positions. His current research interests include design sciences/methodology for high-speed packaging and interconnect and high-speed and low-power chip design, and the application of microelectromechanical machines to electronic systems.