Novel Hardware Architecture for Fast Address Lookups

Pronita Mehrotra, Paul D. Franzon
ECE Department, North Carolina State University,
Box 7911, Raleigh, NC 27695-7911, USA
Ph: +1-919-515-7351, Fax: +1-919-515-2285, email:{pmehrot,paulf}@eos.ncsu.edu

I. Introduction

The most time critical part in packet forwarding is the route lookup which determines the next hop address of the packet. The problem of searching for routes in large databases is compounded by the fact that routing tables store variable length prefixes and their corresponding next hop addresses. In order to forward a packet, routers need to find the longest matching prefix for the destination address. The work presented here describes a new fast and efficient algorithm for searching a large database. The scheme described here 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.

Two main factors motivated this work. The first is the belief that for Gigabit/Terabit routing in the future, performing lookups in hardware will be the only alternative. Larger databases make the problem worse because more memory accesses are required to determine the next hop address. Moreover, a large forwarding table cannot be stored in an on-chip memory or a cache and therefore, very expensive off-chip accesses need to be made. The other important factor is the need for a constant lookup time which is especially important for emerging applications like Optical Burst Switching (OBS) that rely on good estimates of time taken from source to destination. If the set-up time at each node is variable, it would make the delay more unpredictable and would lead to a more inefficient network. Our aim therefore, in designing the forwarding engine was to minimize the off-chip accesses while maintaining a fast, constant lookup time.

The rest of the paper is organized as follows. Section II discusses some of the related work and approaches in performing route lookups. Section III describes our proposed algorithm where only a single off-chip DRAM access is required to determine the next hop address. Section IV discusses some of the details of the hardware implementation and Section V lists some of the results of the scheme. Section VI discusses some of the design issues of the scheme and we finally conclude with Section VII.

II. Related Work

A number of approaches have been used to search for the longest matching prefixes. Most approaches fall under one of the following two methods [1]. In the first method, called a *Search Trie*, each bit in the address is checked and a bit 0 points to the left half of the subtree and a 1 points to the right half of the subtree. The other method, called a *Search Tree* checks the value of the entry with the median value of each subtree. If the value is less than the median value, it is directed to the left half of the subtree and if it is larger, it is pointed to the right half.

The different approaches that use trie based schemes include direct and indirect lookups [2]. The problem with both of these is the large amounts of memory that are required to store the forwarding tables. The number of lookups is small (1-2) but these schemes don't scale well with size. Binary tries store data fairly efficiently. However, they require a large number of memory accesses as compared to the direct or indirect lookup schemes. In the worst case, lookup time can be 32 memory accesses for IPv4 and 128 for IPv6 making this approach unsuitable in the future. Variations of the basic binary trie like Patricia [3] and LC tries[4] improve performance to some extent, though average number of memory accesses is still fairly large. Techniques that use CAMs are not suitable either since CAMs can't be scaled to larger sizes and are more expensive than DRAMs [5].

The other approaches are variations of the binary search. The length of the search in these approaches depends on the number of entries in the routing table. Binary search by itself does not work for the longest matching prefix problem because they can do only exact matches. Lampson et al [6] suggested a modification, where each prefix is encoded as a range. In their scheme, each entry is expanded to two entries, doubling the size of the forwarding table. Waldvogel et al [7] suggested a hash based scheme and a binary search is performed over possible prefix lengths. Their scheme scales well with the size of the routing table and at most 5 hash lookups (for IPv4) are required to determine the next hop address.

Still other approaches taken to improve performance use caching [8], [9]. This may not be very useful for core routers since caching relies on the temporal locality of data and data on core routers exhibit very little temporal locality.

Our Contribution:

Our approach to performing an address lookup attempts to compress the trie information such that it is small enough to make an on-chip implementation possible. By doing this, we reduce expensive off-chip memory accesses to only a single access. To do this, we dissociate the trie data (i.e the next hop addresses) from the pointers and store the data in an off-chip memory. This is not a new concept. For instance, array implementations of trie structures in [10] store the indices of the data array in separate arrays. However, storing indices is equivalent to storing pointers from a hardware perspective. For a routing table with 40,000 entries, a total of over 1Mb of memory is required to store the entire set of indices. Our approach focuses on computing the indices on-the-fly instead of storing them. Since memory speeds are much slower than transistor speeds, by transferring the computation of indices to hardware instead of storing them in memory, we consume less memory without incurring any hits on the overall performance.

III. Description of the Algorithm

This section discusses in detail the working of the proposed algorithm. The proposed scheme compacts the trie data structure such 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.

A. Data Structure

We build the SRAM and the DRAM databases from the conventional multiway trie structure. The offsets in DRAM (equivalent to pointers) are calculated from the bit pattern in the SRAM. The levels in the trie are traversed by computing the offsets for each level. This makes the amount of memory required much less because no pointers (or indices) need to be stored.

The SRAM is built by writing a bit for every node with all its children in the trie structure. Each of the children in the node gives rise to a similar 1 or a 0 depending on the presence or absence of its child nodes. As an example, consider the 4-way trie shown in Figure 1. For this trie, the SRAM contains a "1" for every node with its 4 children. Each 1 in the SRAM bit-pattern, gives rise to 4 more bits in the bit-pattern as shown in Figure 1. A "0" is not propagated while generating the bit-pattern. Also, each "1" in the SRAM corresponds to a row in the off-chip DRAM which stores the 4 possible output port numbers for each of the 4 children. In practice, a DRAM row would hold more next hops and determining the correct DRAM row and column is easy from the bit pattern. This leads to a very compact structure making the SRAM size much smaller than the corresponding DRAM trie structure.

The route lookup is done in two stages. The first stage involves only SRAM lookups and the longest path corresponding to the address is determined from the bit-pattern stored in the SRAM. At the end of this stage, the row and column address of the DRAM where the corresponding next-hop address is stored can be determined. In the next stage, a single DRAM lookup is done and the next-hop address is read. The two stages can be pipelined to give a result every 60-65 ns (random access time for a DRAM) giving over 15 million lookups per second. To improve the speed even further, multiple DRAMS containing identical information can be used in parallel.

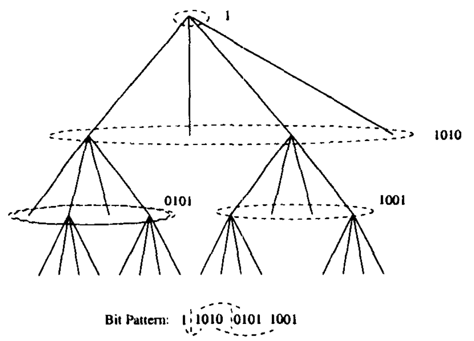

Fig. 1. Sample 4-way trie and the corresponding bit pattern

B. Building the data structure

The data structure to be stored in the SRAM is built from the corresponding binary trie. Our implementation uses a 16-way trie, although any degree of trie can be built. The steps involved in the building the trie are as follows:

Step 1 Each entry from the routing table is read and stored in a list.
Step 2 The list is sorted in an ascending order. For prefixes of different lengths, the prefix with the smaller length is considered to be smaller. For example, 10* would be considered smaller than 100*. The reason for doing this is to ensure that a smaller prefix is always entered first in the trie structure. If the reverse were to happen, additional steps would be required to ensure that correct next-hop entries are stored in child nodes.
Step 3 The root node is created and each of the child node pointers are initialized to NULL.
Step 4 Each entry from the list is read and expanded if necessary to complete the trie. The trie is traversed and the child node pointers and next hop addresses are updated accordingly.

The depth of the trie is $\frac{(No.\ of\ Address\ bits)}{(log_2 X)}$, where X is the degree of the trie. An insertion of an entry into the trie structure can take up to these many lookups 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 cost of building the trie is $N * D$, where D is the depth of the trie.

Once the trie is built, the compact SRAM data structure can be constructed by doing a breadth-first traversal on the trie.

C. Searching the Data Structure

The algorithm to search for the longest matching prefix for a given address is summarized below:
Step 1: The start pointer is initialized to the first X-bit pattern in the SRAM data structure, where X is the degree of the trie.

TABLE I

A SAMPLE DATABASE OF PREFIXES AND THEIR ASSOCIATED HOPS

Prefix	Prefix Length	Next Hop
128.0.0.0	2	3
128.0.0.0	4	6
140.0.0.0	8	3
140.12.0.0	16	2
64.0.0.0	2	7
64.0.0.0	8	12
38.0.0.0	8	5
112.0.0.0	4	9
112.48.0.0	14	5
80.0.0.0	4	2

Step 2: The first $log_2(X)$ bits of the address are read. For a 4-way tree this would mean 2 bits and for a 16-way tree, 4 bits of the address.
Step 3: These address bits are used as the offset in the X bit pattern from the start pointer.
Step 4a: If the bit indicated by the offset is 1, then the start pointer is moved to the next level. The position of the start pointer is calculated by computing the sum of all the previous 1's in the level and multiplying it by X. Steps 2-4 are repeated again.
Step 4b: If the bit indicated by the offset is 0, then the search terminates. The total number of 1's before and including the parent 1 (the 1 that led to the 0) gives the DRAM row number containing the next hop address.

C.1 Example of a Search on a 16-way trie

Table I shows a sample set of prefixes along with their prefix lengths and the next hop addresses. The prefix length is the number of valid bits of the prefix in the routing table. The 16-way trie for the sample set of prefixes is shown in Figure 2. The bit pattern that the SRAM stores for each of the levels is shown in Figure 3. The first few bits of each SRAM row (the "Sum" column) contain the sum of 1's in the current level in previous rows. This is useful to maintain for the following reason. If the SRAM bit-pattern for a particular level spans more than one SRAM row, multiple SRAM accesses would be required to compute the sum of 1's. By adding additional bits in each SRAM row to store this sum, only one SRAM access per level is required. This sum value is easy to compute while generating the SRAM bit-pattern and adds a small overhead to the memory consumption. In this case, since the bit pattern of each level fits in a single SRAM row, these bits are all 0. The off-chip DRAM needs to store only the next hop addresses as shown in Figure 4, where the * represents the default next hop. Each row in the DRAM shown in the figure contains 16 (equal to the degree of the trie) next hop entries. This is only a logical organization and in practice two or more of these rows can be merged together. The row and column address of the DRAM would still be determined easily while traversing the SRAM.

Searching for an address in the SRAM and DRAM strucutres can be easily accomplished by following the steps listed previously.

Sum	Bit Pattern	
0x0000	0x2980	Level 0
0x0000	0x0000 0x0000 0x8000 0x0008	Level 1
0x0000	0x1000 0x80000	Level 2
0x0000	0x0000 0x0000	Level 3

Fig. 3. Bit pattern of the trie as stored in the SRAM

```
* * * * 7 2 7 9 6 3 3 3 * * * * Row 0
* * * * * * 5 * * * * * * * * * Row 1
12 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7Row 2
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 Row 3
6 6 6 6 6 6 6 6 6 6 6 6 3 6 6 6 Row 4
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 Row 5
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 Row 6
5 5 5 5 9 9 9 9 9 9 9 9 9 9 9 9 Row 7
3 3 3 3 3 3 3 3 3 3 3 3 2 3 3 3 Row 8
```

Fig. 4. Data stored in the DRAM

IV. HARDWARE IMPLEMENTATION

The trend in current routers has been to push more intelligence into the line cards to increase message handling capacity [11]. Each of the line cards contains a copy of the forwarding table generated from the main routing table. The forwarding tables in the line cards get updated every couple of seconds. Operations like address lookup, scheduling and configuring the switch fabric are performed by the line cards themselves.

The most time critical part in the design of the router is the forwarding i.e. determining the next hop address from the packet destination address. The block diagram of the forwarding engine is shown in Figure 5. In our implementation, a 16-way trie is used to build the data structure. The bit extractor therefore, picks the next 4 bits of the address. This along with the offset is used to generate a mask for computing the sum of 1's. The sum of 1's 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) and Figure 6 shows the state diagram for traversing the bit-pattern in the SRAM. Each state takes

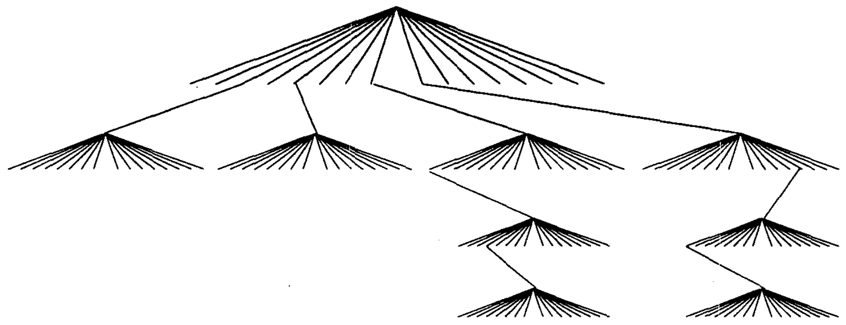
Fig. 2. 16-way trie for the prefixes in Table I

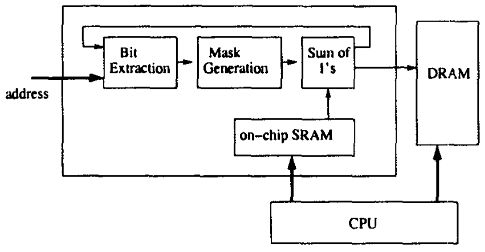

Fig. 5. Block diagram of the Forwarding Engine

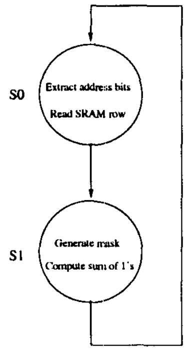

Fig. 6. State Diagram for traversing SRAM

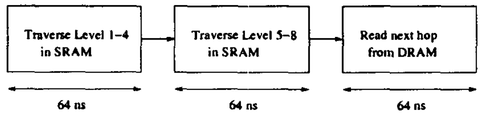

Fig. 7. Pipeline stages of the forwarding engine

8ns to complete and so a total of 16ns 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 (because of 32-bit addresses in IPv4), it would take 128ns to traverse the SRAM. The loop in Figure 6 can be unrolled and pipelined more than once to increase the throughput. In our implementation, we unrolled the loop once and pipelined the two FSMs, to give results every 64ns as shown in Figure 7. This was done to match the random DDR DRAM access time [12]. The main hardware block used in the design of the forwarding engine is the unit that computes the sum of 1's till a given bit position in the SRAM row. The design for this is discussed next.

A. Generating the Mask

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

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 Figure 8. The mask generator is a very simple circuit where the inputs are connected to the outputs as shown in Figure 9. The delay through the generator is the maximum delay at line 127 with a fanout of 128. The 8:128 bit decoder takes around 0.7ns (in 0.18μ technology) to decode [13] while the mask generator again takes around 0.6ns to complete.

B. Computing the sum of 1's

Sum of 1's 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, the adders that would be required is shown in Figure 10. 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 a 0.18μ technology, a 32-bit adder takes about 0.6ns [14]. 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 would be less than 4ns. In all, the total time taken to compute the sum of 1's is well under the budget time of 8ns (for each state in Figure 6).

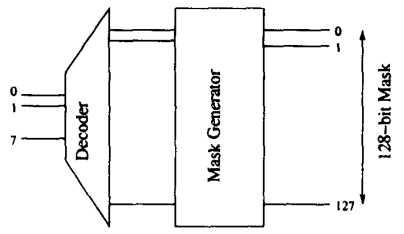

Fig. 8. Generation of Mask from the bit position

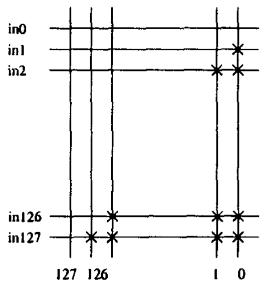

Fig. 9. Mask Generator circuit

V. Performance of the Scheme

We ran the algorithm on practical routing tables from [15]. The results have been summarized in Table II which shows the amount of memory required for these routing tables. For instance, the MaeEast routing table with over 23,000 entries takes around 25KB of SRAM to store the bit pattern and around 12MB of DRAM to store the next hop addresses. In a conventional trie implementation, around 25MB 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.

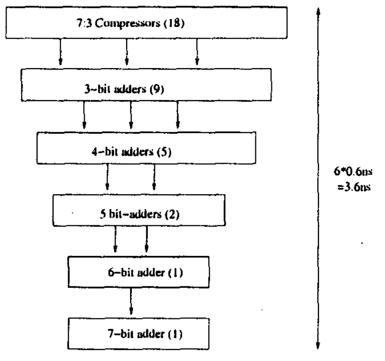

Fig. 10. Adders used in the computation of the sum of 1's

This gives very good scalability which would be very important when routing tables become even larger in the future.

VI. Discussion

The overall compaction achieved in our scheme is much higher than other existing schemes that we are aware of. The required SRAM is small enough (about 35KB 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. In comparison, the forwarding table by Degermark et al [16] uses 5-6 bytes per entry. The implementation by Huang et al [17] has an even larger forwarding 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 in the order of 100ms 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 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 and is common practice in other algorithms.

The number of memory accesses in our implementation, are 8 SRAM accesses and 1 DRAM. The number of SRAM accesses can be reduced further by splitting the SRAM and performing a direct lookup on the first 16 bits. The number of accesses then would be 5 SRAM accesses and 1 DRAM access. This is easily 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 64ns which gives over 15 million lookups per second. In a conventional implementation, the number of memory accesses that would be required are 8 DRAM accesses. DRAM accesses being quite expensive (60-65ns per random read/write as opposed to <10ns for SRAM) [12] the conventional implementation would be much slower than our scheme.

The amount of SRAM compaction can be shown to lie between the following limits:

$1\ bit/entry \leq SRAM\ Memory \leq D.X\ bits/entry$

where, D is the depth of the trie and X is the degree of the trie structure. The upper and lower bounds correspond to extreme cases and are not representative of practical routing tables. The lower bound assumes that the trie is complete whereas the upper bound case assumes that all entries are 32 bits wide (for IPv4) and don't share a common node along the path. This is hardly the case for practical routing tables which are usually sparse and share common nodes.

The overall performance of the forwarding engine, in terms of throughput can be kept constant by altering the hardware FSM. The amount of SRAM memory consumed decreases, mainly due to less wastage in the trie completion

TABLE II

MEMORY REQUIREMENTS FOR VARIOUS ROUTING TABLES

Site	No of Entries	SRAM (KB)	DRAM (MB)	Trie Memory (MB)	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 III

SRAM REQUIREMENTS FOR DIFFERENT DEGREES OF THE TRIE STRUCTURE

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)

step. The amount of SRAM required for various degrees of the trie is shown in Table III. Efficiency of memory consumption increases with decreasing degree of the trie, i.e. fewer bytes per entry are required to store the SRAM data. This is due to less memory wastage in trie completion. The total latency of the address lookup changes with the degree of the trie. This is due to the fact that for smaller degrees, more SRAM accesses have to be made to traverse the trie since the depth of the trie increases. 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 to keep the hardware simple. This comes at a cost of higher memory consumption.

A wider SRAM like 512 or 1024 bit-wide could be used in the design. This would not change the performance of the system but would reduce the memory overhead of the forwarding engine. In the current implementation, 20 bits are used to hold the sum of 1's value for every 128 bits of data in the SRAM row. The memory overhead in the design is an additional 15-16%. By going to a design using 512 bit-wide SRAM, the memory overhead can be reduced to under 4%. The number of memory accesses would still remain the same though additional hardware would be required.

VII. CONCLUSIONS

This paper presents 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 64ns giving a lookup rate of over 15million/sec.

REFERENCES

[1] N. McKeown and B. Prabhakar, "High Performance Switches and Routers: Theory and Practice," in *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," in *Proc. IEEE INFOCOM'98*, (San Francisco, CA), pp. 1382–1391, 1998.

[3] K. Sklower, "A Tree-Based Routing Table for Berkeley Unix," in *Technical Report*, (University of California, Berkeley).

[4] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1083–1092, June 1999.

[5] A. J. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs," in *Proc. IEEE INFOCOM'93*, pp. 1382–1391, 1993.

[6] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups using Multiway and Multicolumn Search," in *Proc. IEEE INFOCOM'98*, vol. 3, (San Francisco, CA), pp. 1248–1256, 1998.

[7] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *Proc. ACM SIGCOMM*, vol. 27, pp. 25–36, Oct. 1997.

[8] T. cker Chiueh and P. Pradhan, "High-Performance IP Routing Table Lookup Using CPU Caching," in *Proc. IEEE INFOCOM'99*, pp. 1421–1428, 1999.

[9] T. cker Chiueh and P. Pradhan, "Cache Memory Design for Network Processors," in *Proceedings of Sxth International Symposium on High-Performance Computer Architecture, 2000*, vol. HPCA-6, pp. 409–418, 2000.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, second ed., 2001.

[11] N. McKeown, "Scalability of IP routers," in *Optical Fiber Communication Conference*, Mar. 2001.

[12] "128Mb DDR SDRAM Datasheet." (http://www.micron.com/products/datasheets/ddrsdramds.html).

[13] L. Mavroidis, "A Low Power 200 MHz Multiported Register File for the Vector-IRAM chip." (http://www.cs.berkeley.edu/ maurog/report.pdf).

[14] B.-H. Lim and J.-K. Kang, "A Self-Timed Pipelined Adder Using Data Align Method," in *The Second IEEE Asia Pacific Conference on ASICs*, pp. 77–80, Aug. 2000.

[15] "Michigan University and Merit Network. Internet Performance Management and Analysis (IPMA) Project." (http://nic.merit.edu/ ipma).

[16] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM*, vol. 27, pp. 3–14, Oct. 1997.

[17] N.-F. Huang and S.-M. Zhao, "A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1093–1104, June 1999.