Searching Very Large Routing Tables in Wide Embedded Memory

Jan van Lunteren

IBM Research, Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland

Abstract— **Exponentially growing routing tables create the need for increasingly storage-efficient lookup schemes that do not compromise on lookup performance and update rates. This paper evaluates the mechanisms that determine the storage efficiency of state-of-the-art IP lookup schemes. A novel scheme named BARTS (Balanced Routing Table Search) is proposed for searching large routing tables in wide embedded memory at OC-192 and OC-768 speeds, while also supporting fast incremental updates. BARTS supports a 38K-entry routing table in 255 KB and a 72K-entry table in 453 KB; a 500K-entry table is estimated to fit into 3 MB. More sophisticated memory management can further reduce these figures to 215, 375 KB, and approx. 2.5 MB, respectively. This is sufficient to handle the large routing tables towards which the Internet seems to be heading in the near future.**

I. INTRODUCTION

The Internet is on the verge of facing a crisis. Its tremendous popularity causes Border Gateway Protocol (BGP) routing tables to grow exponentially, creating serious problems for router manufacturers. Classless Inter-Domain Routing (CIDR) [1], deployed in 1994, brought some relief as it achieved almost linear table growth for several years, as reflected in the Telstra BGP table [2]. However, since 1999, the exponential growth seems to be back at full strength. The largest routing tables today contain about 100K entries. If the current growth continues, routing tables with 500K entries might show up within only a few years.

The problem becomes even more serious because the increase in link speeds driven by advances in optical technology will soon prevent application of SDRAM technology (with its large storage capacity but limited bandwidth) for routing-table lookups. Instead, faster memory technologies have to be used such as SRAM, embedded DRAM, or ternary CAM (TCAM), which are substantially more expensive and provide significantly less storage capacity than SDRAM technology does.

This paper presents a novel IP lookup scheme for searching large routing tables in embedded memory. The Balanced Routing Table Search (BARTS) scheme exploits the wide data buses available in this technology to achieve improved storage efficiency over conventional lookup methods, in combination with wire-speed lookup performance and high update rates.

II. CONVENTIONAL LOOKUP SCHEMES

This section focuses on the mechanisms that affect the storage efficiency of state-of-the-art lookup schemes. The actual performance of these schemes will be discussed in Section V.

IP lookup schemes based on TCAM technology store each routing-table entry in a separate memory location and compare the IP destination address in parallel with all entries [3]. The storage requirements for these schemes grow linear with the routing table size. In contrast, SRAM- and DRAM-based schemes typically process smaller segments of the IP destination address in multiple successive steps. Fig. 1(a) illustrates such a processing step for a tree in which the branching from a parent node p to one of k child nodes $c_1, c_2, .., c_k$ is determined by a so-called *branch function* that takes the marked segment of the search key as operand. Fig. 1(b) shows a common technique to store all child nodes of one parent node in a table, which removes the need to store a separate pointer for each child node. The branch function $f(x)$ provides the offset of the child node within the table.

Popular branch functions are *indexing* (e.g., [4]–[7]) and *testing* (e.g., [8],[9]). Indexing takes the segment value directly as offset. With address segments that correspond to more densely populated parts of the routing table, this is more storage efficient, as it results in better filled tables. With testing, the offset is determined by comparing the address segment against one or multiple test values. Testing is thus more storage efficient for address segments that correspond to more sparsely populated parts of the routing table, as fewer test values have to be stored. Some indexing-based schemes apply a variable address partitioning to obtain segments that correspond to more densely populated parts of the routing table, improving storage efficiency at the cost of a more complex update function.

A different type of "branch function" based on a very efficient encoding of a prefix tree is employed by the Lulea scheme [10], which achieves the highest compression in state-of-the-art lookup schemes, see Section V. Other but less storage-efficient branch functions are used by [11]–[13].

Fig. 2 shows an example of an indexing-based data structure that is obtained by applying prefix expansion [4] on the sample routing table I, for a partitioning of the IP destination address into three segments of 16, 8, and 8 bits. Fig. 2 reveals two other important issues that affect the storage efficiency of a lookup scheme. The first involves the storage and processing of common parts of multiple routing-table entries at one location. Five of the entries in Table I share a common
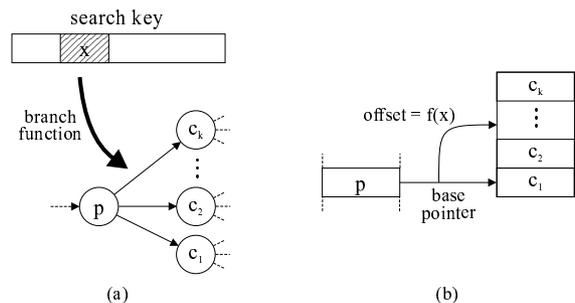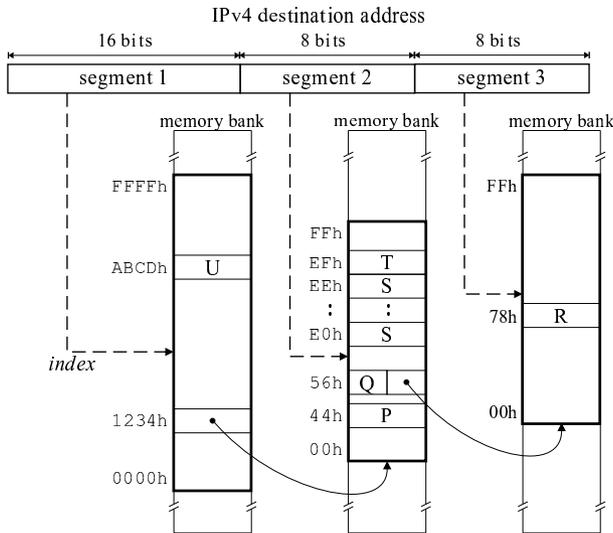

Fig. 1. Branch function.

Fig. 2. Longest matching prefix search using indexing.

TABLE I
SAMPLE ROUTING TABLE.

prefix	length	result	prefix	length	result
123444h	24	P	1234Eh	20	S
123456h	24	Q	1234EFh	24	T
12345678h	32	R	ABCDh	16	U

prefix 1234h, which only uses one table entry in Fig. 2. This mechanism can significantly improve the storage efficiency for large routing tables. TCAM-based schemes, which store each routing-table entry separately, are examples of schemes that do not exploit this mechanism. The second issue deals with so-called *nested* prefixes. Prefix 123456h is a nested prefix of prefix 12345678h, which results in a table entry (at offset 56h in the middle table in Fig. 2) containing both a search result and a pointer. This entry can become rather wide for large routing tables, resulting in inefficient storage usage for table entries that only need to store one of the two fields. Leaf pushing [4] can overcome this problem by moving the next-hop information Q into the empty entries in the table indexed by the third IP address segment. However, this reduces update performance as a larger number of table entries need modification in the case of an update.

Another important issue is memory management. Data structures composed of variable-sized buffers can suffer from memory fragmentation, which can significantly reduce the actual number of routing-table entries that can be stored in a given memory [14]. Memory fragmentation can be reduced by limiting the number of buffer sizes and by defragmentation.

Improving storage efficiency will typically degrade update performance. Lookup performance is not affected if the various levels in the data structure are stored in separate memory banks (as shown in Fig. 2) and if pipelining is used to obtain a lookup rate that only depends on the memory cycle time. Wire-speed lookup performance for OC-192 and OC-768 requires cycle times of at most 26 and 10 ns, respectively, which are

feasible with state-of-the-art embedded memory technologies. Branch functions that are able to efficiently process larger IP address segments have the advantage that fewer memory banks are needed. This typically results in a more efficient use of each bank as it is shared by a larger part of the data structure.

III. BALANCED ROUTING-TABLE SEARCH (BaRTS)

A. Table Compression

The BaRTS scheme is based on a special type of hash function. The hash index consists of a subset of the search key bits that are selected such that the maximum number of collisions for any hash index is limited to a configurable bound N that is a power of 2 and equals at least 2. A collision occurs if two prefixes mapped on the same hash index can both be the longest matching prefix of the search key. Collisions for a given hash index are then resolved by at most N parallel comparisons. Note that the terms *compressed index* and *compressed table* will be used instead of hash index and hash table.

The concept of the compression will now be explained using an example involving the table indexed by the second IP address segment in Fig. 2 for a selected collision bound $N = 2$. This table implements a local prefix search on the prefixes shown in Table II. The compression resolves the problem discussed in Section II regarding a nested prefix with both next-hop information and a pointer by acting as if there are two prefixes, one associated with the next hop information, the other with the pointer. This is the case for prefix 56h in Table II. The corresponding list of prefixes in binary notation now becomes:

    01000100b   (44h)
    01010110b   (56h - next hop Q)
    01010110b   (56h - pointer)
    1110xxxxb   (Eh)
    11101111b   (EFh)

(prefix Eh is padded with x's to match the IP address segment size, where x means "don't care"). The two underlined bit positions are an example of a compressed index, for which the maximum number of prefix collisions is limited to $N = 2$ (Section III-B will discuss how these bit positions can be determined). Fig. 3 shows the corresponding compressed table. The prefixes that are mapped on a certain compressed-index value are included in the corresponding compressed table entry as tuples consisting of a test value, a test mask and a search result. The set bits in the test mask indicate which bit positions are covered by the prefix. Note that prefix Eh is mapped on multiple compressed-index values.

A compressed table entry will be called a *block*, and parameter N will be called the block size. A tuple containing next-hop information will be denoted as a next-hop entry, and a tuple containing a pointer will be called a pointer entry. The bits of the IP address segment that form the compressed index can be specified by a so-called *index mask* that is stored together with the pointer to the compressed table as is shown in Fig. 3. The index mask specifies the hash function that is used to compress the table. The actual entry formats will be defined in Section III-C.

1616

prefix	length	result
44h	8	P
56h	8	Q, ptr
Eh	4	S
EFh	8	T
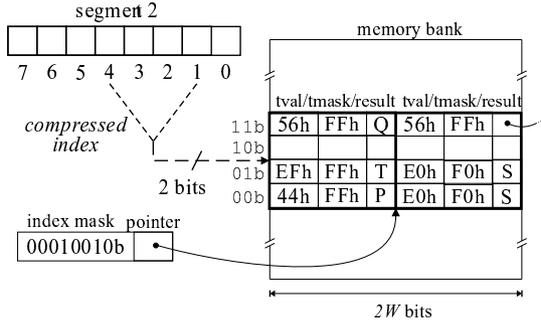
Fig. 3. Compressed table for $N = 2$.

N	counter "size"	total number of counters				
		$s = 4$	$s = 5$	$s = 6$	$s = 7$	$s = 8$
2	2 bits	81	243	729	2187	6561
4	3 bits	65	211	665	2059	6305
8	4 bits	33	131	473	1611	5281
16	5 bits	9	51	233	939	3489
32	6 bits	-	11	73	379	1697

counter overflow. This counter can be realized by a register with $\log(N) + 1$ bits. Table III shows the required counter "size" and number of counters as a function of block size N and IP address segment size s.

All counters are connected by combinatorial logic to a common bus on which the prefixes are written as test values and test masks in successive cycles. The combinatorial logic causes the counter to increase by one if the prefix maps on the compressed-index value corresponding to that counter. If the counter value exceeds the block size, an overflow flag is raised. Arbitration logic will determine the smallest compressed index for which none of the corresponding counters has experienced an overflow. Fig. 4 shows the counter and corresponding combinatorial logic for a compressed index specified by an index mask `01100100b` and with value `101b`.

The index-mask calculation can be further optimized by filtering out nested prefixes that can never be the longest matching prefix for a given compressed-index value (this will not be discussed further in this paper).
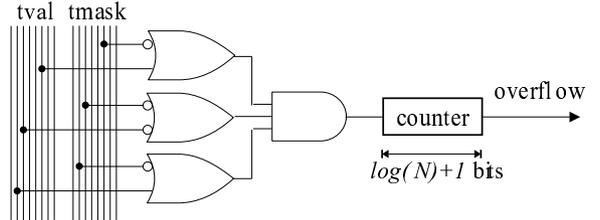
The memory width allows an entire block to be read in one access. Parallel test logic will then determine the longest matching next-hop entry (i.e., the matching next-hop entry with the most set bits in the test mask) as well as the longest matching pointer entry. The test logic can be simplified as there can be only one matching pointer entry because pointer entries involve an exact match over the entire IP address segment. A further simplification of the parallel test logic can be realized by ordering the next-hop entries by their prefix length.

B. Compressed-Index Calculation

This section presents a hardware-based compressed-index "calculation" for optimum compression of a table involving k prefixes in exactly k cycles. A similar calculation can be implemented in software, but will perform slower.

An optimum compressed index will be calculated by a "brute-force" count of the actual number of collisions that occur for each possible value of each possible compressed index. The smallest compressed index for which the number of collisions for all values is bounded by N is then selected as the optimum compressed index. This requires a counter array that includes one counter for every possible combination of a compressed index and a compressed-index value. The total number of counters can be derived as follows. A total of $\binom{s}{k}$ different compressed indices consisting of k bits can be extracted from an IP address segment with s bits, each of which can have 2^k different values. It can be shown that a compressed index can consist of a maximum of $s - \log(N) + 1$ bits for a block size N (derivation omitted owing to space limitations). This results in a total number of counters equal to

$$T = \sum_{k=0}^{s-\log(N)+1} \binom{s}{k} \times 2^k . \tag{1}$$

A counter needs to be able to count from 0 to $N + 1$ for a block size N, where the maximum counter value corresponds to a

Fig. 4. Combinatorial logic and counter for index mask `01100100b` and compressed-index value `101b`.

C. Data Structure

Fig. 5 shows the entries used to build the actual data structure. The various entry types are distinguished by the first two bits. Entry type `00` is an empty entry. Entry type `01` is a next-hop entry containing a test value, a test mask, and next-hop information as discussed in Section III-A. Entry type `10` is a pointer entry. As this type of entry involves an exact match over the entire IP address segment, no test mask is needed. Instead, an index mask is included, defining the compressed index used to index the compressed table that is referenced.

Entry type `11` is a special pointer entry involving an index mask equal to zero for referring to compressed tables consisting of at most N prefixes that are stored within a single block. To efficiently handle small tables containing fewer than N prefixes, multiple tables can be stored in the same block. The entries of each compressed table are identified by an entry-selection field, which is stored with the pointer instead of the
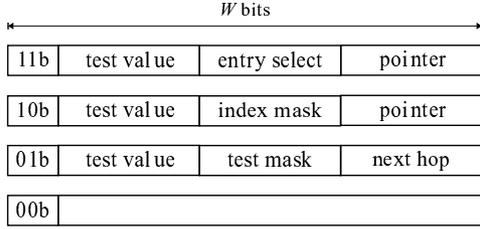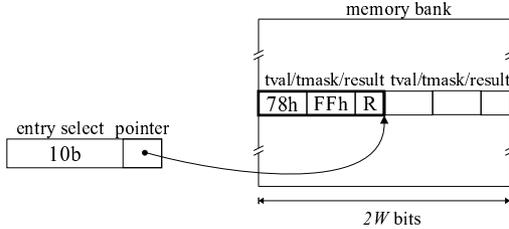
Fig. 5. Entry format definition.

Fig. 6. Entry selection field.

index mask. Entry selection can be based on an offset together with a count, or on a bit vector in which each bit corresponds to a location within the block. An example of the latter is shown in Fig. 6 for compressing the table indexed by the third IP address segment in Fig. 2. The entry-selection field and memory management can be simplified by enforcing power-of-2 table sizes through padding with empty entries (type `00`).

D. Incremental Updates and Memory Management

The data structure can be incrementally updated by creating modified copies of the corresponding compressed tables in memory, and linking them by an atomic write operation into the structure similar as described in [4]. Hardware-based compressed-index calculation and compressed-table construction enable update rates in excess of hundreds of thousands per second, as will be discussed elsewhere [15].

If power-of-2 table sizes are enforced (as described above), then the memory manager has to support $s + 2$ buffer sizes corresponding to compressed tables with $2^0, 2^1, .., 2^{s+1}$ entries, which matches very well with a buddy system [16]. Otherwise, all buffer sizes corresponding to small compressed tables with fewer than N entries have to be supported as well, requiring more sophisticated memory management. If memory fragmentation were to become a problem, then the BARTS scheme can be adapted to use fewer buffer sizes by either reducing the segment size s, or by only using compressed indices that correspond to supported buffer sizes. In the latter case, a suboptimum compression will be achieved, but the implementation of the index-mask calculation as described in Section III-B becomes simpler because fewer counters are needed.

IV. SIMULATIONS

Several simulations have been performed with the routing tables listed in Table IV, which are made publicly available by the IPMA project [17]. These simulations involved block sizes between 2 and 32 in combination with the following partitions of the IP address: 16 8 8, 16 4 4 8, 14 5 5 8, and 12 6 6 8 (the

notation 16 8 8 represents a partition involving three segments of 16, 8, and 8 bits). The first segment of each partition indexes an uncompressed table, the remaining segments "index" compressed tables. All entries fit into 32 bits for the simulated routing tables, while also including an 18-bit next-hop field. Fig. 7 shows the effect of the address partition on the storage requirements for a fixed block size of 8, which corresponds to a memory width of 32 bytes. Fig. 8 shows the effect of the block size (memory width) for a fixed partition 12 6 6 8. All simulations involved power-of-2 buffer sizes, except two simulations with partition 12 6 6 8 and block sizes 16 and 32, marked in Fig. 8 as 'N=16*' and 'N=32*'.

Partition 12 6 6 8 performs best of all simulated partitions. It allows the 72K-entry paix table to fit into 492 KB for $N = 8$,

TABLE IV
IPv4 ROUTING TABLES (FEBRUARY 2000).

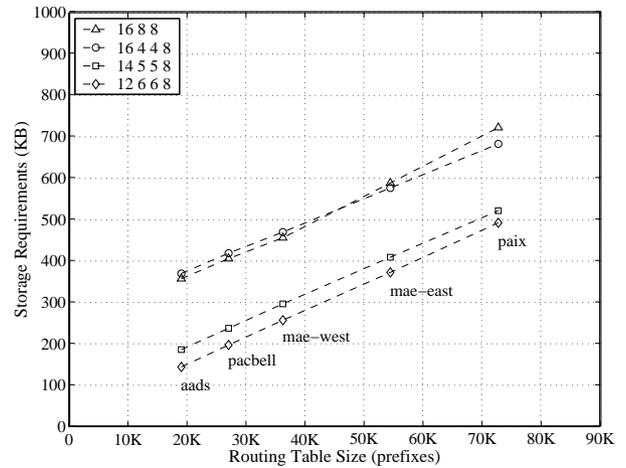| routing table	prefixes	routing table	prefixes
Aads	19056	Mae-east	54499
PacBell	27085	Paix	72825
Mae-west	36292		

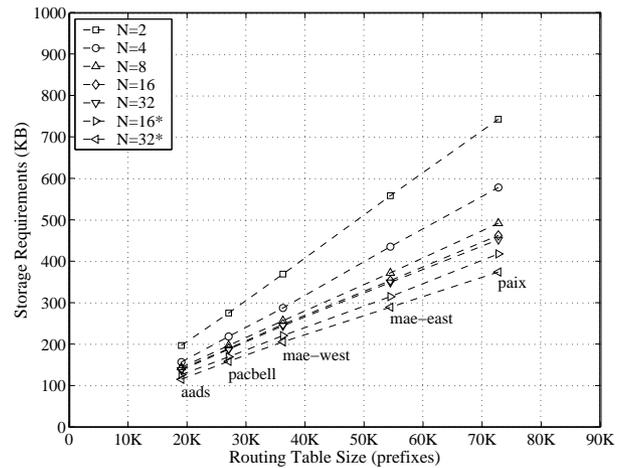

Fig. 7. Effect of address partition for $N = 8$.

Fig. 8. Effect of block size (memory width) for partition 12 6 6 8.

1618

into 463 KB for $N = 16$, and into 453 KB for $N = 3$, using power-of-2 buffer sizes. The same table fits into 419 KB for $N = 16$ and into 375 KB for $N = 3$ if non-power-of-2 buffer sizes are supported. Storage efficiency improves not only for larger block sizes, but also for larger routing tables. For example, the average number of bytes needed to store one routing-table entry decreases from 7.4 bytes for aads to 6.3 bytes for paix for a partition 12 6 6 8 and $N = 3$ (power-of-2 buffer sizes). This results from the increasing number of common parts between routing-table entries, see Section II. A conservative estimate, assuming the average storage per entry will remain at 6.3 bytes for routing tables larger than paix, implies that a 500K-entry routing table will fit into only 3 MB.

V. COMPARISON

Table V lists reported and estimated storage requirements of several well-known lookup schemes for a popular 38K-entry routing table (most of these are from [18]). It also gives estimated storage requirements of BARTS for this table size, obtained from the simulation results by interpolation.

BARTS achieves the second-best performance. Only the Lulea scheme performs better as it needs only 160 KB with an address partition of three segments. However, compared to BARTS, the Lulea scheme has three major drawbacks: (1) It needs up to four memory accesses to process a single IP address segment. BARTS needs exactly one. (2) It seems unable to support fast incremental updates. Moreover, updating a chunk (term used in [10]) always requires full recompression. BARTS supports fast incremental updates and performs inserts directly if there are empty entries in a block (which is likely for larger block sizes). (3) With the Lulea scheme, compressed chunks can have 256 different non-power-of-2 sizes (mainly due to the varying number of pointers) for an address segment of 8 bits. Incremental modifications of the data structure would require complex memory management, while the large number of buffer sizes could lead to significant memory fragmentation. BARTS achieves good performance with only $s + 2$ power-of-2 buffer sizes for an address segment of s bits, and can efficiently be adapted to use fewer buffer sizes.

BARTS outperforms the other lookup schemes, which are mostly based on fixed "branch functions" involving indexing or testing, thanks to its *adaptive* compression: more bits are used for indexing to compress densely populated tables efficiently, but most bits are only involved in testing to compress sparse tables. BARTS also outperforms lookup schemes based on ternary CAMs for large routing tables. BARTS uses an average of 6.3 bytes to store a single entry of the 72K-entry paix table (partition 12 6 6 8, $N = 3$). Removing the 18-bit next-hop information from those 6.3 bytes leaves about 32 bits that are effectively used to "store" and "search" the prefix part of each routing-table entry. These 32 binary bits require less chip area than the 32 associative ternary bits a ternary CAM uses to "store" and "search" the prefix part of a routing table entry.

VI. CONCLUSIONS

BARTS is a novel IP lookup scheme that achieves wire-speed lookup performance for OC-192 and beyond using pipelining techniques, that efficiently handles large routing tables with more than 100 K entries in wide embedded memory and supports fast incremental updates using a hardware-based update function. This paper has focused primarily on the storage requirements. Another paper [15] will analyze the update performance and memory management issues in more detail.

TABLE V
STORAGE REQUIREMENTS (IN KB) FOR A 38K-ENTRY ROUTING TABLE.

Algorithm	Storage	Algorithm	Storage
Patricia (BSD)	3262	BARTS (12 6 6 8, N=8)	269
Binary Search[11]	1600	BARTS (12 6 6 8, N=16)	259
6-way search[8]	950	BARTS (12 6 6 8, N=32)	255
LC Trie[6]	700	BARTS (12 6 6 8, N=16*)	231
Prefix Expansion[4]	450	BARTS (12 6 6 8, N=32*)	215
Lulea[10]	160		

REFERENCES

[1] Y. Rekhter and T. Li, "An architecture for IP address allocation with CIDR," RFC 1518, September 1993.
[2] BGP Table, http://telstra.net/ops/bgptable.html.
[3] D. Shah and P. Gupta, "Fast updates on ternary-CAMs for packet lookups and classification," *Proc. Hot Interconnects VIII*, August 2000, pp. 145-153.
[4] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Computer Sys.*, vol. 17, no. 1, pp. 1-40, February 1999.
[5] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," *Proc. IEEE INFOCOM*, vol. 3, pp. 1240-1247, April 1998.
[6] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1083-1092, June 1999.
[7] N.-F. Huang and S.-M. Zhao, "A novel IP-routing lookup scheme and hardware architecture for multigigabit switching routers," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1093-1104, June 1999.
[8] 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.
[9] P. Gupta, P. Prabhakar, and S. Boyd, "Near optimal routing lookups with bounded worst case performance," *Proc. IEEE INFOCOM*, vol. 3, pp. 1184-1192, March 2000.
[10] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink, "Small forwarding tables for fast routing lookups," *Computer Commun. Rev.*, vol. 27, no. 4, pp. 3-14, October 1997.
[11] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," *Computer Commun. Rev.*, vol. 27, no. 4, pp. 25-36, October 1997.
[12] D. Yu, B.C. Smith, and B. Wei, "Forwarding engine for fast routing lookups and updates," *Proc. IEEE Globecom*, vol. 2, December 1999, pp. 1556-1564.
[13] W-S.E. Chen, C-T.J. Tsai, "A fast and scalable IP lookup scheme for high-speed networks," *Proc. IEEE Int'l Conf. Networks*, September 1999, pp. 211-218.
[14] S. Sikka and G. Varghese, "Memory efficient state lookups with fast updates," *Computer Commun. Rev.*, vol. 30, no. 4, pp. 335-347, October 2000.
[15] J. van Lunteren, "Scalable IP lookups for OC-192 and beyond," unpublished.
[16] J.L. Peterson and T.A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, pp. 421-431, June 1977.
[17] Michigan University and Merit Network, Internet Performance Measurement and Analysis (IPMA) Project, http://www.merit.edu/ipma.
[18] P. Gupta, "Routing lookups and packet classification: theory and practice," Tutorial at Hot Interconnects VIII, Stanford, August 2000, http://klamath.stanford.edu/~pankaj/talks/.