# Searching Very Large Routing Tables in Fast SRAM

Authors: Jan van Lunteren

IBM Zurich Research Laboratory
Säumerstrasse 4
CH-8803 Rüschlikon, Switzerland
jvl@zurich.ibm.com

Published in: Proceedings of the 10th IEEE International Conference on Computer Communications and Networks ICCCN'01, pp. 4-11, Scottsdale, Arizona, October 2001.

© 2001 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

# Searching Very Large Routing Tables in Fast SRAM

Jan van Lunteren
IBM Research
Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
jvl@zurich.ibm.com

*Abstract*— **The Balanced Routing Table search (BARTs) scheme is a novel method for searching large routing tables at OC-192 and OC-768 speeds. The scheme employs a compression technique that leaves all prefix information intact and creates a stand-alone data structure that can be incrementally updated. Performance-critical parts of the update operation are suitable for hardware implementation, enabling update rates beyond several hundred thousand per second. Simulations with actual routing tables have shown that BARTs supports a routing table with 36,292 entries in only 292 KB of storage, a table with 72,825 entries in 555 KB, and a future table with 500,000 entries is estimated to fit in 3.7 MB.**

## I. Introduction

Advances in optical technology will soon boost the link speeds in the Internet backbone from OC-48 to OC-192 and beyond. The next generation routers, although still operating in the electrical domain, will have to keep pace with the link speeds in order to fully exploit the developments in optical technology, as this seems to be the only way to handle the exponentially growing Internet traffic volumes. IP lookup is one of the performance-critical router functions that requires full attention in this context.

IP lookup determines for each incoming packet the next hop to which it will be forwarded by performing a longest-matching prefix search on a routing table using the packet's IP destination address. Although many IP lookup schemes have been published already (see Section II), two recent developments create the need for new schemes that are significantly more storage efficient than conventional schemes, without compromising on lookup and update performance.

The first development is the exponential growth of the BGP routing tables, caused by the tremendous popularity of the Internet. Classless Inter-Domain Routing (CIDR) [1] achieved almost linear table growth for several years after its deployment in 1994, but was unable to sustain this. The exponential growth seems to be back at full strength since 1999 [2]. While the largest routing tables today contain about 100,000 entries, tables with 500,000 entries might become reality within only a few years if the current growth continues.

The second development is that the increase in link speeds 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.

This paper presents a novel IP lookup scheme called BARTs (Balanced Routing Table search) which exploits a new compression technique to achieve improved storage efficiency over conventional lookup methods, in combination with wire-speed lookup performance and high update rates. The presented scheme is optimized for SRAM technology. A second version of the BARTs scheme optimized for wide embedded memory is presented in [3]. The paper is organized in the following way: Section II analyzes state-of-the-art lookup schemes; Section III introduces the BARTs scheme; Section IV presents simulation results, which are compared with the performance of existing schemes in Section V; Section VI concludes the paper.

## II. Conventional Lookup Schemes

This section focuses on the mechanisms that affect the storage efficiency of state-of-the-art IP lookup schemes. Actual performance figures will be discussed and compared in Section V.

TCAM-based lookup schemes store each routing table entry in a separate memory location, and compare those fully in parallel with the IP destination address in a single cycle [4]. The storage requirements for these schemes grow linearly with the size of the routing table. 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 trie. 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. This eliminates 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.

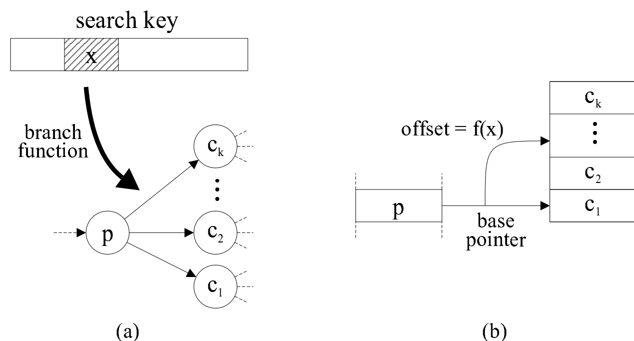


Fig. 1. Branch function.

Popular branch functions are *indexing* (e.g., [5-8]) and *testing* (e.g., [9, 10]). Indexing uses the segment value directly as offset into the table. This is more storage efficient for address segments that correspond to densely populated parts of the routing table, as this results in a better fill rate of the tables. Testing, on the other hand, determines an offset by comparing the address segment against one or multiple test values. Testing is more storage efficient for address segments corresponding to sparsely populated parts of the routing table, as this requires fewer test values 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. This typically improves the storage efficiency at the cost of a more complex update function.

The Lulea scheme employs a different "branch function," which is based on a very efficient encoding of a prefix tree [11]. This scheme achieves the highest compression in state-of-the-art lookup schemes as will be discussed in Section V. Other but less storage-efficient branch functions are used in [12-14].

Fig. 2 shows an example of an indexing-based data structure obtained by applying prefix expansion [5] on the routing table in Fig. 3 (with prefixes in hexadecimal notation) for a partitioning of the IP destination address into three segments of 16, 8, and 8 bits. Fig. 2 reveals two important issues that affect the storage efficiency of a lookup scheme. The first issue involves the storage and processing of common parts of multiple routing table entries at one location. Four of the routing table entries in Fig. 3 share a common prefix 1234h, which uses only 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 prefixes 1234566h, 12345678h, and 123456CDh, which results in a table entry (at offset 56h in the lower table indexed by the second address segment in Fig. 2) that contains both a search result and a pointer. This entry can become rather wide for large routing tables, resulting in an inefficient storage usage for table entries that only need to store one of two fields. Leaf pushing [5] can overcome this problem by moving the next hop information R into the empty entries in the table indexed by the third address segment. However, this reduces update performance because a larger number of table entries will need modification in the case of an update.

The last 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 [15]. Lookup schemes that require fewer different buffer sizes will suffer less from memory fragmentation.

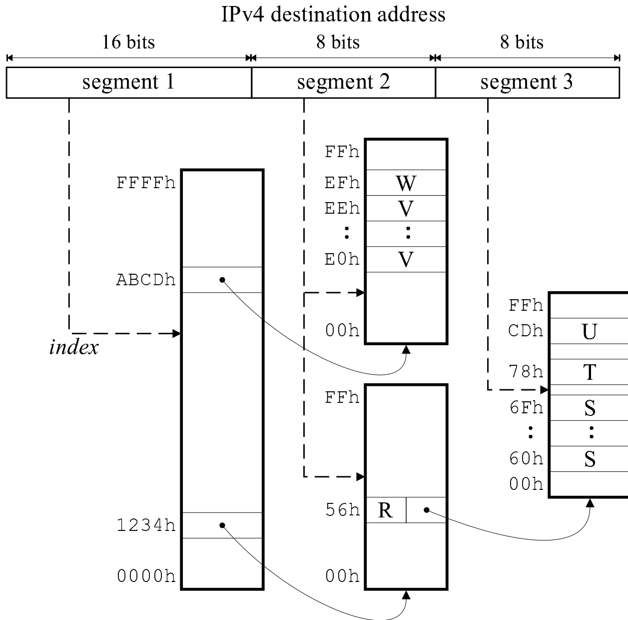


Fig. 2. Longest-matching prefix search using indexing.

| prefix | length | result | prefix | length | result |
|---|---|---|---|---|---|
| 123456h | 24 | R | 123456CDh | 32 | U |
| 1234566h | 28 | S | ABCDEh | 20 | V |
| 12345678h | 32 | T | ABCDEFh | 24 | W |

Fig. 3. Sample routing table.

## III. BALANCED ROUTING TABLE SEARCH (BARTS)

### A. Table Compression

The BARTs scheme for fast SRAM is based on a perfect hash function for longest-matching prefix searches. The hash index consists of a subset of the search key bits selected such that for each hash index at most one prefix exists that can be the longest matching prefix of the search key. The terms *compressed index* and *compressed table* will be used instead of hash index and hash table. The concept of the compression will be explained using an example that involves the table indexed by the third IP address segment in Fig. 2, and one that involves the upper table indexed by the second IP address segment in the same figure. These two tables implement "local" prefix searches on the prefixes listed in Figs. 4(a) and (b).

Fig. 4(a) contains the following prefixes, written in binary notation:

0110xxxxb (6h)
01111000b (78h)
11001101b (CDh)

| prefix | length | result |
|---|---|---|
| 6h | 4 | S |
| 78h | 8 | T |
| CDh | 8 | U |

(a)

| prefix | length | result |
|---|---|---|
| Eh | 4 | V |
| EFh | 8 | W |

(b)

Fig. 4. Local prefixes.

(prefix 6h is padded with *x*'s to match the IP address segment size; *x* means don't care). The two underlined bit positions have the property that each prefix differs from any other prefix in at least one of the two positions. These two bits can therefore be used as a compressed index, for which no collisions occur. This means that for each possible value of the compressed index, at most one prefix exists that can be the longest matching prefix of the search key. If such a prefix exists for a certain compressed index value, then this prefix is said to be "mapped" on that value. Fig. 5 shows the corresponding compressed table. The prefixes mapped on the compressed index values are included in the corresponding table entries 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 part of the prefix. The bits of the IP address segment forming 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. 5. 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.

Prefix Eh is a nested prefix of prefix EFh in Fig. 4(b). In this case there are no bit positions in which the two prefixes are different from each other. To avoid collisions, the compressed index now has to cover the bit positions that are only part of the longer prefix and not of the shorter. These bit positions are underlined in the following binary notation of the two prefixes:

`1110xxxxb` (Eh)
`11101111b` (EFh)

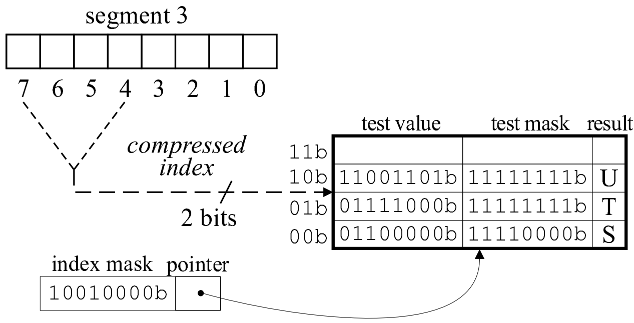


Fig. 5. Compressed table.

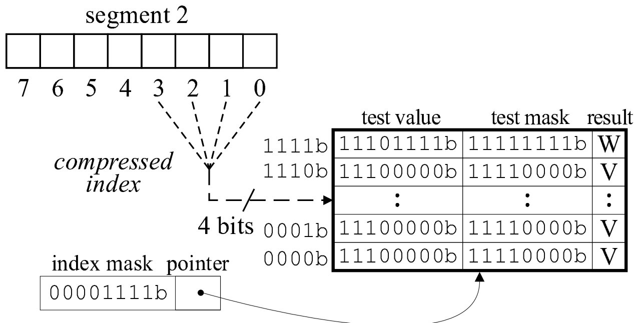


Fig. 6. Compressed table containing nested prefixes.

If the address segment contains a value `1111b` at the underlined bit positions, then both prefixes might match but only prefix `11101111b` can be the *longest* matching prefix. For any other value, only prefix `1110b` can be matching. Fig. 6 shows the corresponding compressed table. This figure shows that one prefix can be mapped on multiple compressed index values.

### *B. Compressed Index Calculation*

An optimum compressed index is the smallest compressed index that is collision-free, because this results in the largest compression. A collision-free compressed index has the following two properties (see the preceding section): (1) For each pair of prefixes that are not nested, the compressed index contains at least one bit position in which the two prefixes are different. (2) For each pair of nested prefixes, the compressed index includes all bit positions that are part of the longer prefix but not part of the shorter prefix. The first property avoids collisions between prefixes that are not nested, whereas the second avoids collisions between nested prefixes.

The bit positions at which two prefixes differ from each other can be determined by calculating a bit-wise exclusive-or (XOR) product over the bit positions that are part of both prefixes. The differing bit positions show up as set bits in the XOR product. If prefixes are represented as test values ($tval$) and test masks ($tmask$), then a bit-wise XOR product $x_{ij}$ between a prefix $i$ and a prefix $j$ over the bit positions that are part of both prefixes can be calculated according to:

$$x_{ij} = (tval_i \otimes tval_j) \wedge (tmask_i \wedge tmask_j) \, . \tag{1}$$

($\otimes$ and $\wedge$ represent bit-wise XOR and AND operators). If a compressed index has Property 1 then this implies that the corresponding index mask shares at least one set bit position with each non-zero XOR product $x_{ij}$:

$$\forall_{x_{ij} \neq 0} \; (imask \wedge x_{ij}) \neq \texttt{00...0b} \, . \tag{2}$$

The XOR product $x_{ij}$ of two nested prefixes $i$ and $j$ contains zero bits only. The bit positions that are only part of the longer but not of the shorter of two nested prefixes can be determined by calculating a bit-wise XOR product $y_{ij}$ over the test masks of these prefixes according to:

$$y_{ij} = (tmask_i \otimes tmask_j) \, . \tag{3}$$

If a compressed index has Property 2 then this implies that the corresponding index mask includes all set bit positions that occur in each XOR product $y_{ij}$ calculated for any pair of nested prefixes:

$$\forall_{x_{ij} = 0} \; (imask \wedge y_{ij}) = y_{ij} \; , \tag{4}$$

which can be rewritten as

$$(Y \wedge imask) = Y, \quad Y = \bigvee_{x_{ij}=0} y_{ij} \, . \tag{5}$$

| | | | |
|---|---|---|---|
| $tval_1$ | = 0110xxxxb | $tmask_1$ | = 11110000b |
| $tval_2$ | = 01111000b | $tmask_2$ | = 11111111b |
| $tval_3$ | = 11001101b | $tmask_3$ | = 11111111b |
| $x_{12}$ | = 00010000b | | |
| $x_{13}$ | = 10100000b | | |
| $x_{23}$ | = 10110101b | $Y$ | = 00000000b |
| $imask$ | = 10010000b | | |

(a)

| | | | |
|---|---|---|---|
| $tval_1$ | = 1110xxxxb | $tmask_1$ | = 11110000b |
| $tval_2$ | = 11101111b | $tmask_2$ | = 11111111b |
| $x_{12}$ | = 00000000b | $Y = y_{12}$ | = 00001111b |
| | | $imask$ | = 00001111b |

(b)

Fig. 7. XOR products for a given set of test values and test masks.

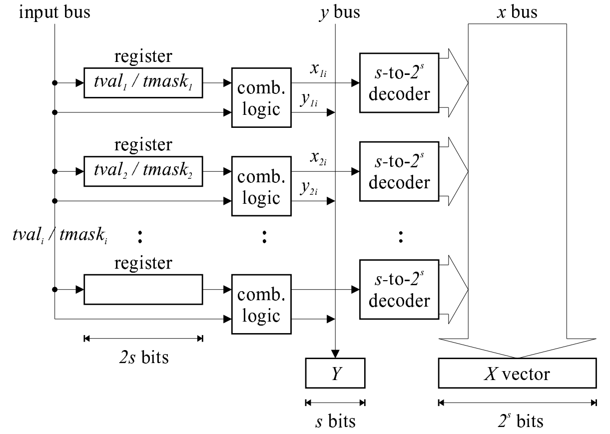


Fig. 8. Parallel XOR product calculation.

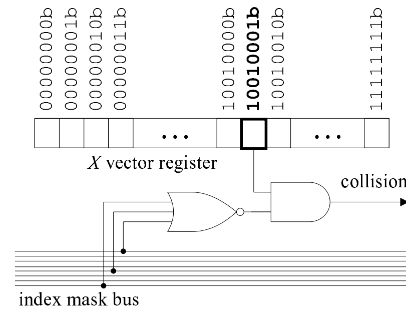


Fig. 9. Parallel index mask testing.

($\vee$ represents a bit-wise OR operator). Figs. 7(a) and (b) show the various test values, test masks, XOR products $x_{ij}$ and $y_{ij}$ and index masks corresponding to the prefixes shown in Figs. 4(a) and (b). The index masks in Figs. 7(a) and (b) share at least one set bit position with each non-zero XOR product $x_{ij}$ (2), and include all set bit positions that occur in $Y$ (5).

To test whether an index mask has Property 1 by using (2) requires all (different) XOR products $x_{ij}$ that are calculated for all pairs of non-nested prefixes. For $m$ prefixes a total of $\binom{m}{2}$ XOR products have to be calculated. Storing and processing these XOR products as separate bit vectors is rather inefficient. Instead, a single bit vector of $2^s$ bits will be used, containing one bit for each possible XOR product that can exist for an address segment consisting of $s$ bits. This vector will be called $X$ vector. If a certain XOR product $x_{ij}$ is calculated for a pair of non-nested prefixes, than the corresponding bit in the $X$ vector will be set.

To test whether an index mask has Property 2 by using (5) only requires the bit-wise OR product $Y$ of all XOR products $y_{ij}$ instead of the individual products. For this purpose it is sufficient to store only the $Y$ bit vector, consisting of the same number of $s$ bits as the IP address segment, and update this vector each time a XOR product $y_{ij}$ is calculated for a pair of nested prefixes.

Fig. 8 shows a hardware-based XOR product "calculator" that implements these concepts and can calculate all XOR products for $m$ prefixes in exactly $m$ cycles. The prefixes are written in successive cycles over the input bus into a set of registers as pairs of test values and test masks. Control logic ensures that each new prefix is written into an empty register. While a new prefix is being written over the input bus, the XOR products between the new prefix and the prefixes that are already stored in the register set are calculated fully in parallel (this is implemented by combinatorial logic). The $x$ and $y$ buses, both implementing a logical bit-wise OR function, in combination with some decoder logic, are used to update the $Y$ and $X$ vectors for the resulting XOR products $x_{ij}$ and $y_{ij}$ as described above.

An optimum compressed index can now be determined by generating all possible index masks ordered according to an increasing number of set bits, and testing these against (2) and (5) for the calculated XOR products. The first index mask that satisfies both conditions corresponds to the smallest compressed index that is collision-free. Simple combinatorial logic can test in a single cycle whether an index mask shares at least one set bit position with each non-zero XOR product for which the corresponding bit is set in the $X$ vector. Fig. 9 shows the required logic for an XOR product `10010001b`. If only index masks are generated that satisfy (5) then an optimum compressed index can be determined in at most $2^s$ cycles because a total of $2^s$ different index masks exist for an address segment consisting of $s$ bits. The speed of the index mask calculation can be increased by generating index masks that already have a minimum number of bits set based on the number of prefixes. The index mask calculation can also be adapted to the memory manager operation by only evaluating index masks that correspond to supported buffer sizes (Section III-E).

### C. Data Structure

Fig. 10 shows a complete data structure consisting of compressed versions of the tables shown in Fig. 2. The table indexed by the first address segment has a fixed (maximum) size

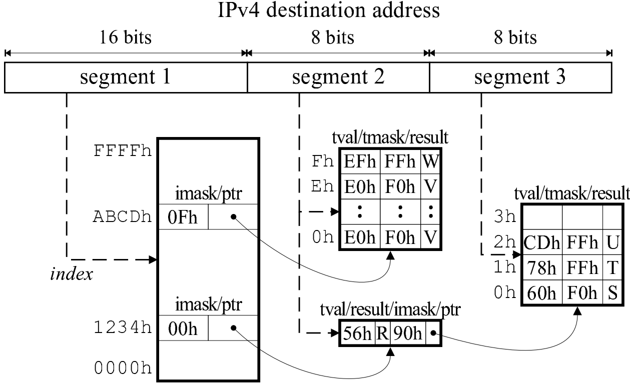


Fig. 10. Compressed data structure.

of $2^{16} = 64$ K entries, that is not changed by recompression in case of updates. This concept of a table that is "fully" indexed by a relatively large first address segment is also employed in several other schemes. As the routing table portion corresponding to the first address segment will be densely populated for many routing tables, not much can be gained by compression. At the same time, it allows a large part of the IP destination address to be processed in one step, leaving fewer bits for further examination.

Fig. 11 shows the formats of the various types of entries, which are distinguished by the first two bits. Entry type `00` is an empty entry. Entry type `01` consists of a test value, a test mask, and next hop information. Entry type `10` contains a pointer. As this entry type involves an exact match over the entire address segment, no test mask is needed. Instead an index mask is included that defines the compressed index used to index the referenced compressed table.

Entry type `11` is a special entry for nested prefixes. It includes both next hop information and a pointer. This entry type occurs, for example, in the lower table in Fig. 10, which is indexed by the second IP address segment. Because the next hop information and the pointer fields can become significantly wide for large routing tables, it was decided to split this entry type into two entries that are linked to each other as shown in Fig. 11. The next hop information is distributed over

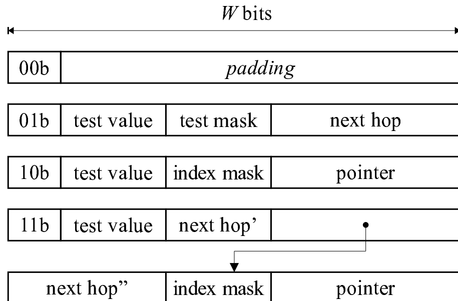


Fig. 11. Entry formats.

both entries. This allows a smaller uniform entry width than if both fields were stored in one entry. This is more efficient as less storage is needed for padding, but at the cost of an additional memory access each time this entry type is accessed during the lookup process. For a given routing table, the maximum number of memory accesses $b$ needed for one lookup is in the range

$$c \leq b \leq 2c - 1 \quad (6)$$

for an IP destination address partitioned in $c$ segments. The lower bound corresponds to the situation that the data structure contains no entries of type `11`. The upper bound corresponds to the situation that the data structure contains at least one lookup "path" with $c - 1$ entries of type `11` and one entry of type `01`.

It is possible to convert an entry of type `11` into an entry type `10` by inserting the next hop information related to the nested prefix into the referenced compressed table at the next level as a local prefix of length zero. This can be regarded as a form of leaf pushing [5]. Applying this concept on entries of type `11` that are in a "critical path," allows one to force a certain value of $b$ anywhere within the above range. However, this will typically increase the storage requirements and decrease update performance as described in Section II (leaf pushing).

### D. *Incremental Updates and Stand-Alone Capability*

A routing table update involving the insertion of a new or the removal of an existing prefix requires the modification and recompression of at most one compressed table containing multiple entries, and the corresponding addition or removal of a sequence of single-entry tables "linked" to that multi-entry table. Thus, at most one compressed index calculation is required for each update operation. Updates can be performed incrementally by first creating new tables and modified copies of existing tables in memory, and then linking these by an atomic write operation into the existing data structure. If the lookup and update processes access the memory using a form of time-division multiplexing, then updates can be performed without interrupting the lookup process. The corresponding memory accesses can be interleaved in an arbitrary order.

The necessary modifications of the data structure for an update operation can be determined using a second data structure. However, the BARTs scheme can also be used in a stand-alone fashion without a second data structure, because all prefix information is available in the form of test values and test masks. By performing a lookup operation using the prefix involved in the update operation as a search key, it is possible to determine all tables that need modification.

The maximum number of memory accesses needed for an update when the data structure is operated in a stand-alone fashion occurs in the following cases:

- A compressed table with $2^s$ entries has to be recompressed into a compressed table with $2^{s-1}$ entries for a delete operation, or vice versa for an insert operation, where $s$ is the size of the largest IP address segment. This requires a total

of $2^s + 2^{s-1}$ memory accesses for reading the original table and writing the modified copy of the table.

- A sequence of $c - 1$ single entry tables have to be linked to the above compressed table for an insert operation, involving $c-2$ entries of type `11` and one entry of type `01`, where $c$ is the number of segments into which the IP address is partitioned. This requires a total of $2(c-2)+1 = 2c-3$ memory write accesses.

Added to these access counts are (a) the maximum number of memory read accesses for a lookup on the prefix involved in the update to determine the tables needing modification (6) and (b) a memory write access to link new and modified copies of tables into the data structure, so that the following maximum number of memory read and write accesses is obtained for performing an update operation:

$$2^s + 2^{s-1} + 4c - 3. \quad (7)$$

Routing tables can include prefixes that cannot become the longest matching prefix of the search key. An example is a prefix `0b`, which is fully covered by two prefixes `00b` and `01b`. Fully covered prefixes do not appear in uncompressed tables such as those shown in Fig. 2, nor do they appear in the corresponding compressed tables. Because these prefixes are relevant for the update process when the data structure is operated in a stand-alone fashion, they can be added specifically by including them in empty table entries (entry type `00`). As routing tables typically contain very few of these prefixes, this will not be discussed further.

### *E. Memory Management*

An IP address segment consisting of $s$ bits can index compressed tables of $s+1$ different sizes, all being powers of 2, containing between one and $2^s$ entries. To achieve optimum compression, the memory manager has to support all corresponding buffer sizes. Such a memory manager could for example be based on a buddy system [16], which matches very well with the power-of-2 buffer sizes. 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 a smaller set of 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 and faster because fewer index masks have to be evaluated.

## IV. SIMULATIONS

Several simulations have been performed with the routing tables listed in Fig. 12 that are made publicly available by the IPMA project [17]. These involved the IP destination address partitions listed in Fig. 13 (the notation 16 8 8 represents a partition having three segments of 16, 8, and 8 bits). The address partitions have been chosen such that only segments that cover the bulk of prefixes with lengths up to 24 are varied. The table

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

Fig. 12. IPv4 routing tables (February 2000).

| stand-alone data structure | | | not stand-alone data structure | | |
|---|---|---|---|---|---|
| number of segments | partition | entry width | number of segments | partition | entry width |
| 4 | 8 8 8 8 | 36 bits | 3 | 16 8 8 | 36 bits |
| 5 | 8 6 5 5 8 | 32 bits | 4 | 14 5 5 8 | 32 bits |
| 6 | 8 4 4 4 4 8 | 32 bits | 5 | 12 4 4 4 8 | 32 bits |

Fig. 13. Simulated IP destination address partitions.

indexed by the first segment of each partition has a fixed size, as described in Section III-C. Fig. 13 also shows the entry widths used with the various partitions.

### *A. Storage Requirements*

Fig. 14 shows the storage requirements for the simulated partitions. Partition 8 4 4 4 4 8 performs best, closely followed by partition 12 4 4 4 8. Partition 8 4 4 4 4 8 allows the mae-west table with 36,292 entries to fit in 292 KB and the paix table with 72,825 entries to fit in 555 KB. Partitions 16 8 8 and 8 8 8 8 clearly perform worse than the other partitions.

The storage efficiency of BARTs improves for larger routing tables. For example, the average number of bytes needed to store a routing table entry decreases from 9.1 bytes for aads to 7.9 bytes for paix for a partition 12 4 4 4 8. This is due to

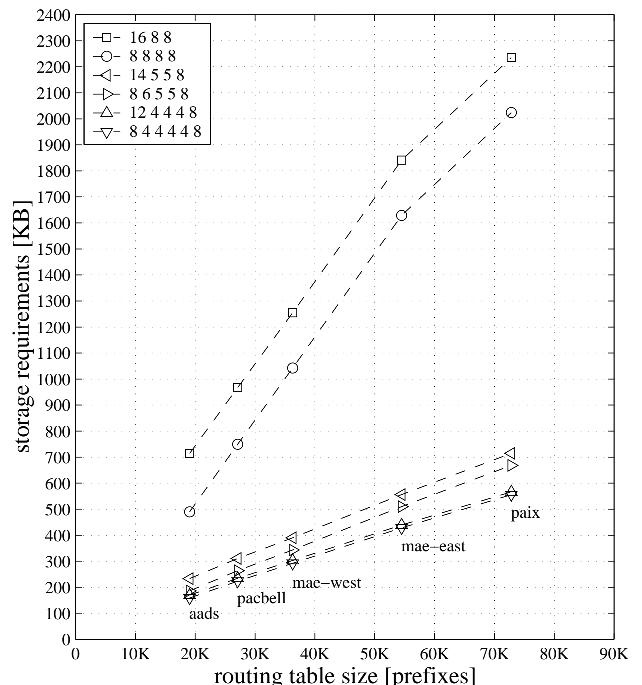


Fig. 14. Storage requirements.

the increasing number of common parts between routing table entries (see Section II). A conservative estimate based on the assumption that the average storage per entry will remain at 7.9 bytes for routing tables larger than paix implies that a future routing table containing 500,000 entries will be able to fit in about 3.7 MB.

### B. Lookup and Update Performance (Stand-Alone Operation)

Wire-speed processing of minimum-sized packets requires 26 M lookups/second for OC-192 and 100 M lookups/second for OC-768. If BARTs is used in a stand-alone fashion, and the lookup and update processes access the memory using a form of time-division multiplexing, then the wire-speed lookup rates in combination with the maximum number of memory accesses per lookup determine the memory bandwidth that should be allocated to the lookup process. The remaining memory bandwidth in combination with the maximum number of memory accesses per update determine the worst-case update performance that can be achieved for the given memory system (assuming that the compressed index calculation and new table construction are performed in parallel).

Fig. 15 lists the worst-case update rates that can be achieved in combination with wire-speed OC-192 lookup speed on a single bank of SRAM with a cycle time between 4 and 6 ns for the three "stand-alone" partitions. Fig. 16 shows the same for OC-768 lookup speed and a pipelined operation of BARTs on four banks of SRAM. Both figures also show the maximum numbers of memory accesses used to calculate the update rates. The maximum number of memory accesses per lookup $b$ appeared to be dependent on the number of address segments $c$ according to

$$b = c + 2 \quad (8)$$

which implies that the simulated data structures contained at most two nested prefixes in a single search path (Section III-C). The maximum number of memory accesses per update were calculated using (7). This calculation did not consider the fixed-sized table indexed by the first segment of an address partition, as it is not involved in a regular update operation. Tables indexed by the last segment were not considered either, because the simulated routing tables contained a negligible number of prefixes with lengths greater than 24 that cover the last address segment.

Average lookup and update rates can be significantly higher than the worst-case numbers presented in Figs. 15 and 16. Although some papers present average rates as actual performance figures, this is considered here as misleading as these average rates can only be determined for given (often artificial) traces of IP packets and routing table updates, and are not generally applicable.

| | maximum number of memory accesses | | worst-case number of updates/second | | |
|---|---|---|---|---|---|
| partition | lookup | update | $t_c$ = 6ns | $t_c$ = 5ns | $t_c$ = 4ns |
| 8 8 8 8 | 6 | 397 | 27 K/s | 111 K/s | 237 K/s |
| 8 6 5 5 8 | 7 | 113 | - | 159 K/s | 602 K/s |
| 8 4 4 4 4 8 | 8 | 45 | - | - | 933 K/s |

Fig. 15. Worst-case update rates for wire-speed OC-192 lookup performance (26 M lookups/s) on a single SRAM bank with a cycle time $t_c$.

| | maximum number of memory accesses | | worst-case number of updates/second | | |
|---|---|---|---|---|---|
| partition | lookup | update | $t_c$ = 6ns | $t_c$ = 5ns | $t_c$ = 4ns |
| 8 8 8 8 | 6 | 397 | 168 K/s | 504 K/s | 1.01 M/s |
| 8 6 5 5 8 | 7 | 113 | - | 885 K/s | 2.65 M/s |
| 8 4 4 4 4 8 | 8 | 45 | - | - | 4.44 M/s |

Fig. 16. Worst-case update rates for wire-speed OC-768 lookup performance (100 M lookups/s) on four SRAM banks with a cycle time $t_c$.

## V. COMPARISON

Fig. 17 shows reported and estimated storage requirements of several well-known lookup schemes for a popular 38K-entry routing table (most of these are obtained from [18]) as well as estimated storage requirements of BARTs for a table of this size, obtained by interpolating the simulation results of the four best-performing address partitions in Fig. 13.

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 four memory accesses to process a single IP address segment, whereas BARTs needs typically one, and at most two. (2) It seems unable to support fast incremental updates. Furthermore, updating a chunk (term used in [11]) always requires full recompression. BARTs supports fast incremental updates and performs inserts directly if there are empty locations in a compressed table. (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 eight bits. Incremental modifications of the data structure would require complex memory management, whereas the large number of buffer sizes could lead to significant memory fragmentation. BARTs achieves good performance with only $s + 1$ power-of-2 buffer sizes for an address segment of $s$ bits, and can efficiently be adapted to use fewer buffer sizes.

The BARTs scheme outperforms all 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 efficiently compress densely populated tables, whereas most bits are only involved in testing to compress sparsely populated tables.

| Algorithm | Storage | Algorithm | Storage |
|---|---|---|---|
| Patricia (BSD) | 3262 KB | BARTs (14 5 5 8) | 408 KB |
| Binary Search[12] | 1600 KB | BARTs (8 6 5 5 8) | 359 KB |
| 6-way search[9] | 950 KB | BARTs (12 4 4 4 8) | 317 KB |
| LC Trie[7] | 700 KB | BARTs (8 4 4 4 4 8) | 305 KB |
| Prefix Expansion[5] | 450 KB | | |
| Lulea[11] | 160 KB | | |

Fig. 17. Storage requirements for a 38K-entry routing table.

## VI. CONCLUSIONS

The BARTs scheme allows to effectively balance lookup performance, storage requirements, update performance and memory management complexity for a given memory technology. This is achieved by an adaptive table compression technique, which efficiently compresses densely populated tables as well as sparsely populated tables. Simulations have shown that an actual routing table with 36,292 entries fits in only 292 KB of storage, and a table with 72,825 entries fits in 555 KB. A future table with 500,000 entries is estimated to fit in 3.7 MB. Only Pink has reported a smaller data structure, which, however, does not support incremental updates [11].

Performance-critical parts of the update function can be implemented in dedicated hardware to achieve very high update rates. Applications that do not require fast update performance can implement the update operation in software. A key feature of the compression is that it leaves the prefix information intact, which enables a stand-alone data structure that can be used for both lookup and update operations. Performance evaluations have shown that wire-speed lookup performance in combination with update rates beyond several hundred thousand per second are feasible using a single bank of SRAM for OC-192, and four banks for OC-768.

## 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] J. van Lunteren, "Searching very large routing tables in wide embedded memory," unpublished.

[4] D. Shah and P. Gupta, "Fast updates on ternary-CAMs for packet lookups and classification," *Proceedings of Hot Interconnects VIII*, August 2000, pp. 145-153.

[5] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Computer Systems*, vol. 17, no. 1, February 1999, pp. 1-40.

[6] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," *Proceedings IEEE INFOCOM*, vol. 3, April 1998, pp. 1240-1247.

[7] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, June 1999, pp. 1083-1092.

[8] 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, June 1999, pp. 1093-1104.

[9] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *IEEE/ACM Trans. Networking*, vol. 7, no. 3, June 1999, pp. 324-334.

[10] P. Gupta, P. Prabhakar, and S. Boyd, "Near optimal routing lookups with bounded worst case performance," *Proceedings IEEE INFOCOM*, vol. 3, March 2000, pp. 1184-1192.

[11] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink, "Small forwarding tables for fast routing lookups," *Proceedings of ACM SIGCOMM*, in *Computer Commun. Rev.*, vol. 27, no. 4, October 1997, pp. 3-14.

[12] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," *Proceedings of ACM SIGCOMM*, in *Computer Commun. Rev.*, vol. 27, no. 4, October 1997, pp. 25-36.

[13] D. Yu, B.C. Smith, and B. Wei, "Forwarding engine for fast routing lookups and updates," *Proceedings IEEE Globecom*, vol. 2, December 1999, pp. 1556-1564.

[14] W-S.E. Chen, C-T.J. Tsai, "A fast and scalable IP lookup scheme for high-speed networks," *Proceedings IEEE Int'l Conf. on Networks*, September 1999, pp. 211-218.

[15] S. Sikka and G. Varghese, "Memory efficient state lookups with fast updates," *Proceedings of ACM SIGCOMM*, in *Computer Commun. Rev.*, vol. 30, no. 4, October 2000, pp. 335-347.

[16] J.L. Peterson and T.A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, June 1977, pp. 421-431.

[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/.