A memory-efficient scheme for Address Lookup using Compact Prefix Tries

Anand Sarda and Arunabha Sen
Department of Computer Science and Engineering,
Arizona State University, Tempe, AZ 85287
Email: {asarda, asen}@asu.edu

***Abstract*— In this paper we present a new memory-efficient scheme for address lookup that exploits the caching support provided by general-purpose processors. We propose *Compact Prefix Tries*, in which prefixes occurring at multiple levels of a subtrie are compressed into a single node that fits in a single cache line. The scheme performs well in compressing dense as well as sparse tries. For an IP core router (Mae-West) database with 93354 prefixes, the simulation results for Compact Prefix Tries show up to 70% improvement in lookup performance and up to 33% reduction in memory when compared with LC-$Tries$. In fact, the entire forwarding table for Mae-West required only 829 KB space. Measurements for Compact Prefix Tries, when compared with most existing schemes, show better results in terms of memory usage as well as lookup speeds. Moreover, as the memory usage is significantly less and sparse tries with long paths can be compressed into only a few nodes, this scheme is particularly attractive for IPv6.**

I. IP ADDRESS LOOKUP PROBLEM

The primary role of a router is to route the packet to its destination. In order to do so, for each packet it receives, the router must determine the address of the next hop where it should be forwarded. Router maintains a table called forwarding table that stores the forwarding information. Each entry in the routing table has a network address, length and an output port identifier or next hop address. The pair of address and its length is called as a *prefix*. When a packet is received, the router extracts the destination address from the packet header. It is then matched with the prefixes in the routing table using some lookup algorithm to find the next hop address. This operation is called as *address lookup*. Since the prefixes are of different lengths in the router tables, multiple prefixes match a given address. So, in order to find the next hop address for the destination address, the router has to find the most specific prefix or the *longest matching prefix*. The router then forwards the packet from incoming port to corresponding outgoing port. This is called as *switching*.

II. PREVIOUS WORK

Several algorithms for efficient prefix matching lookups have been presented in technical literature in recent years. The classical solution for IP address lookup is using tries. A trie [5] is a tree-based data structure in which prefixes are organized on digital basis using the bits of prefixes to decide the branching. Another method used for compressing the height of trie is Level Compression [10][11]. In this technique, for any given prefix length, dense areas with common ancestor are aggregated into a single 2^k-ary branching node. This scheme maintains a good balance of memory usage, search speed and update times. Another trie compression scheme in hardware is presented in [4]. Multibit tries are also used for compressing levels in a trie. Multibit tries [2] [14] [1] speedup the lookup speed of tries by inspecting many bits simultaneously. A technique for expanding and compressing multibit tries is presented in [2]. The Lulea scheme [3] compresses multibit trie nodes to reduce storage to fit in the cache. In the worst-case, $O(W)$ memory accesses are required, but these accesses are to fast cache memory. Controlled Prefix Expansion [14] optimizes the multibit trie using dynamic programming. The scheme reduces memory, improves performance and the authors claim it to be very tunable.

III. COMPACT PREFIX TRIES

This section will present the new scheme designed for compressing tries for address lookups. While designing the data structure for this scheme, the primary goals were to reduce the memory required and memory accesses. Both these goals go hand in hand. Reducing the number of memory accesses is important because they are relatively slow as compared to processor speeds and are usually the bottleneck of lookup procedures. Reducing the size of data structure allows the data structure to fit entirely in the cache memory. This means that accessing the data will be extremely fast as compared to accessing it from relatively slow main memory (DRAM/SDRAM).

Even if the entire forwarding table dose not fit into cache, it is beneficial to group correlated information together, so that a large fraction of it will reside in cache. Other important factor is locality of reference. Locality observed in traffic patterns will keep the most frequently used pieces of the data structure in cache, so that most lookups will be fast.

A. Basic Idea

Path compression reduces the height of trie by compressing single child nodes. This only works if the trie structure is sparse. Level Compression reduces the height by a significant factor, but is more effective in the denser areas of the trie. Multibit tries work in a similar way as Level Compressed tries, but prefixes of intermediate length have to be expanded and they require exponential memory 2^k, where k is the length by which the prefix is expanded.

 0-7803-7974-8/03/$17.00

The basic idea of Compact Prefix Tries is to group the prefixes occurring at multiple levels into a single compressed node that can fit into a cache line. Instead of using the prefix expansion like in multibit tries, they are expanded using Boundary Prefix Expansion [13]. Only the boundary ranges of a prefix are stored. The advantage of doing this is that if a prefix has to be expanded by length, only 2 entries are required instead of 2^k, as in case of multibit tries. This scheme was originally proposed in [7]. For matching a prefix inside the compressed node, a binary search is performed on the set of Boundary Expanded prefixes.

The Figure 1 (a) shows a Trie as a large triangle. The maximum height of the trie is equal to maximum levels it has, usually the size of destination address. This trie is then partitioned into smaller subtries of different heights, such that each subtrie fits into a node equal to the size of cache line. These nodes shall be called as *Compressed SubTrie Nodes* or *CSTnodes*. The number of levels a subtrie covers is called as *stride*. A subtrie contains *prefixes* as well as *links*. A *link* points to a subtrie at the next level. All prefixes in a subtrie are expanded and then compressed along with links into a single CSTnode. The Compact Prefix Trie can be viewed as a tree of n-ary CSTnodes which are nothing but compressed subtries. After all the levels are compressed the Trie structure looks as shown in the Figure 1 (b).

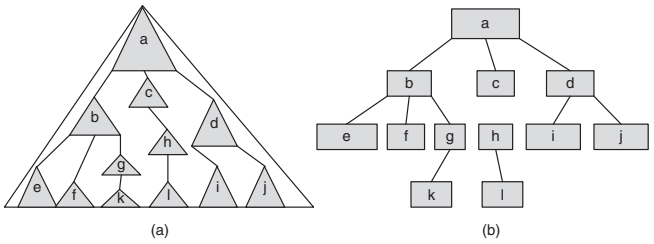

Fig. 1. Compressed Trie after Partitioning

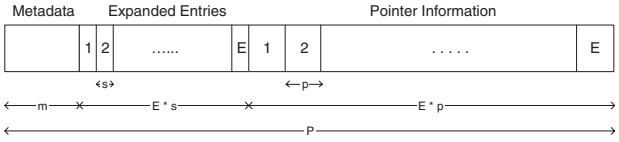

Fig. 2. CSTnode Structure

B. Node Structure

A CSTnode is used to store a subtrie. Each CSTnode has 3 parts as shown in Figure 2. The first part is called Metadata part, where the information like, type of node, number of entries, stride of the subtrie is stored. m bytes are required to store Metadata. Second and third parts are arrays of length E, where E is the maximum entries that can be stored in a CSTnode. In the second part expanded entries are stored. Each entry is of size s bytes. The maximum length of the expanded entry that can be stored in this CSTnode is equal to number of bits in s. If s = 1 byte, prefix of maximum length 8 can be stored as an expanded entry. A key is searched in these entries by performing binary search. The third part stores the pointers to next hop table and nodes at the next level. Size required to store pointers is p bytes. The elements in second and third part are mapped 1:1. For every expanded entry in second part, the corresponding pointer information is stored in the third part. Once the binary search on entries stops, the corresponding pointers in third part are used to find the next hop and node at next level. Total size of the CSTnode is denoted by P.

C. Compressing a Trie into Compact Prefix Trie

Like path, level compressed or multibit tries, Compact Prefix Tries can be created by first building a binary trie and then compressing it. The four steps involved in compressing the trie are described below.

1) Start from the root node and find level l, such that the subtrie up to level l will fit into a CSTnode.
2) Expand the prefixes in the subtrie using Boundary Prefix Expansion.
3) Fill the node after compacting boundary expanded prefixes.
4) For each subtrie below level l repeat step 1.

Algorithms used at each step are explained below.

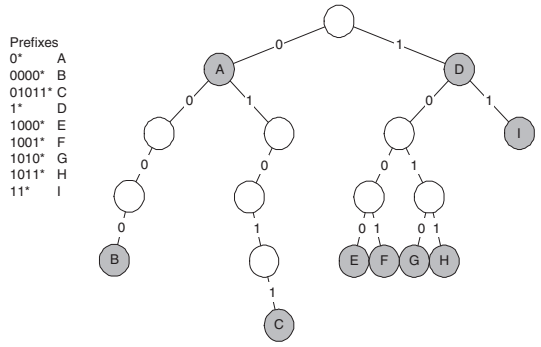

Fig. 3. Sample Trie

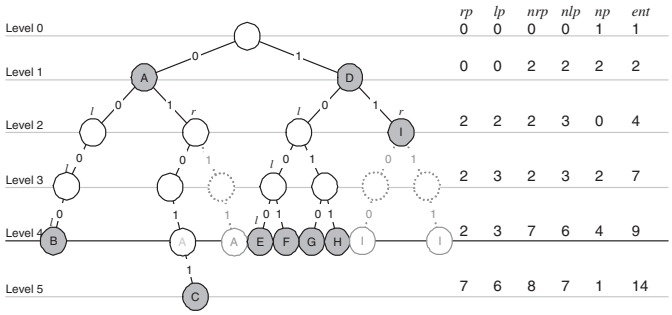

Fig. 4. Finding Exact Levels

1) Algorithm to Find Exact Levels: We present an algorithm that finds the exact number of levels that can be compressed. Five variables are maintained as the search proceeds in Breadth first manner. The variables rp and lp count the number of right paths (high endpoint of a range) and left paths (low endpoint of a range) originating from prefixes at previous levels. For

example, in Figure 4 all the nodes from level 2, for path 0000, belong to the prefix A's (0*) left path. nrp and lrp count the right and left paths for the next level. Links or prefixes that do not belong to any right or left path are counted by np. Let ENT be the number of entries found at each level that can be stored into a CSTnode. Flag r or l is used to indicate if the node is part of right path or left path for a prefix. In the above example, nodes at paths 00, 000 and 0000 are marked as l indicating that they belong to left path for prefix A. The algorithm is for finding exact number of levels is given below.

procedure $FindingExactLevels()$

```
1: unmark all the nodes in the trie and start breadth first search from the
   root node
2: lvl ← 0, rp ← 0, lp ← 0, nrp ← 0, nlp ← 0 and ENT ← 0
3: while ENT ≤ max do
4:    np ← 0
5:    for each node v in the subtrie at level l do
6:       if v is a prefix then
7:          if v is unmarked then
8:             nrp ← nrp + 1
9:             nlp ← nlp + 1
10:            np ← np + 1
11:         else
12:            if mark(v) = r then
13:               nlp ← nlp + 1
14:            else if mark(v) = l then
15:               nrp ← nrp + 1
16:            end if
17:         end if
18:         if v has children then
19:            mark left child as l and right child as r
20:         end if
21:      else
22:         if v is unmarked then
23:            np ← np + 1
24:         else
25:            if mark(v) = r then
26:               mark right child as r
27:            else if mark(v) = l then
28:               mark left child as l
29:            end if
30:         end if
31:      end if
32:   end for
33:   ENT ← rp + lp + np
34:   lvl ← lvl + 1
35:   rp ← nrp
36:   lp ← nlp
37: end while
38: return lvl − 1
```

The value $rp + lp + np$ calculates the exact number of entries that will be stored in the CSTnode for level lvl. This is done by adding all the right and left paths originating from previous levels (rp and lp) and the links (np) that point the subtries below level lvl. Since this algorithm keeps track of the extended paths of all the prefixes at intermediate levels, it finds the exact number of entries at each level that can fit into one node. Also, as Breadth First search is used, each node in the subtrie is accessed only once. Hence the complexity for a subtrie with N nodes is $O(N)$.

2) Extracting Entries: Once the number of levels that can be compressed is known, the next step is to extract and expand the prefixes occurring at intermediate levels. Figure 5 shows the spans for prefixes in the sample trie from Figure 3. Span for prefix A is from 0000 to 0111. Prefix D spans from 1000 to 1111 and I from 1100 to 1111. But since prefix I is of larger length than D, the destination address falling into the range 1100 and 1111 will find 'I' as the longest matching prefix. While extracting the entries, care has to be taken to preserve the spans of each prefix, as well as mark the Start and End for each span. Depth first search can be used for finding and expanding the prefixes. The algorithm is as follows:

1) Start from the root node. Follow Depth first search towards left.
2) If a node is not marked, mark the node as 'visited' and do
   a) If the length of the prefix is equal to the stride length, store its bit string and mark it as 'Point Range' (Prefixes B, E, F, G and H).
   b) Else, if the node is a valid prefix, expand its prefix by appending 0's, store and mark the bit string as 'Low range' (Prefix A, D and I).
3) If the node is already marked as 'visited'
   a) If the node's height is equal to the stride length and it is not a valid prefix, then store the bit string for that node and mark it as a 'Link' (Node at path 0101).
   b) If it is a valid prefix, expand it by appending 1's and mark the string as 'High Range' (Prefixes A, I and D).

After all the nodes in the subtrie are traversed, a sorted array of bit strings is obtained with the each bit string marked as either Low Range, High Range, Point Range or Link. The bit strings extracted from the sample trie are shown in Figure 5. The markers High Range (Hi) and Low Range (Lo) are used to determine the span of a prefix.

Since Depth First Search is used, again the time complexity is $O(N)$.

3) Filling Nodes: Now, the expanded set of bit strings in Figure 5 should be compacted by removing the duplicate strings. The strings are processed one by one, from lowest value to the highest, in a similar way as described in [7]. Finally the compacted entries look like in Figure 6.

The Type indicates the maximum stride length allowed for this CSTnode. As we process at most $2N$ entries, the complexity for this operation is also $O(N)$. The size of the CSTnode is dependent on number of entries after compacting, and is never greater than the maximum allowed size. The node manager, which manages the creation, alignment, assignment and maintenance of the nodes, is requested for a CSTnode of the particular size. When a free CSTnode is assigned, the metadata, prefix strings and the information about next hop and next node pointers are filled into it.

D. Extensions and Optimizations

Some extensions and optimizations can be applied the scheme to increase the lookup speed and to make it more suitable for IPv6. These are explained below.

1) Initial array for 16 bits: It is possible to reduce the number of memory accessing by having a table for first 16 bits. The size of this table will be 2^{16}, i.e. 65535 entries. The 16 bits can be used to index into the array. Each entry in the array stores the corresponding longest matching prefix and a link to the nodes of Compact Prefix Tries is also stored. Since indexing into the initial array requires just one memory access, the total number of accesses required for lookup is reduced.

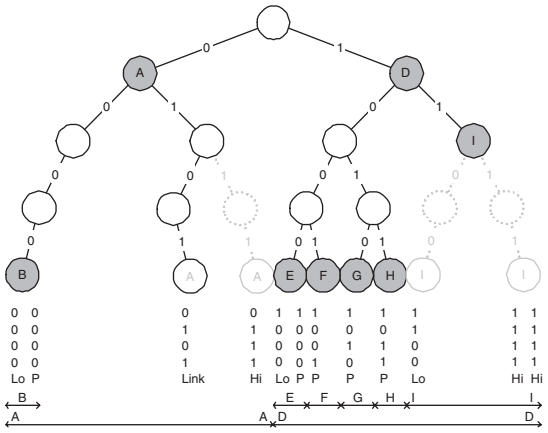

Fig. 5. Extracting Prefixes from Levels

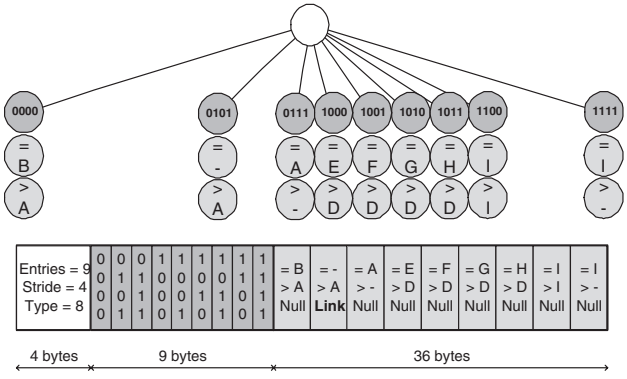

Fig. 6. Node after filling the entries

2) Increasing the size of CSTnodes: Increasing the size CSTnodes reduces the depth of the tree. But cache line accesses required for lookup are also increased. It is possible to reduce the maximum number of cache line accesses by intelligently arranging the data inside the node. The bit strings require less space and hence requires less cache lines for storing. The accesses for binary search will be within cache lines containing the bit strings only. If all the bit string cache lines are available in cache, only one memory access per node is required for retrieving pointer information. Thus, even if the size of the complete data structure is larger than the size of total cache available, the scheme will perform better if a small part of the structure (Metadata and bit string entries) is in cache.

E. Larger stride lengths

In some cases when the tries are very sparse, it might happen that even though the maximum allowed length is reached, the node might be not be completely filled. In these cases, it is possible to store more levels in same CSTNode by decreasing the total entries that can be stored. The advantage of using larger stride length is that more number of levels can fit into a node. If the trie is sparse, this reduces the average length of the trie, thus reducing average number of memory accesses required. This is especially useful for IPv6 where the length of destination address is 128 bits and tries are sparse. The disadvantage is that the branching inside the lookup function increases.

F. Searching

The Search procedure for finding the best matching prefix is described below. The same procedure can be used for both IPv4 and IPv6 prefixes.

1) If the Initial array is present, index into the it using the first 16 bits of the destination address.
2) If the pointer to CSTnode at next level is null, return the next hop pointer stored at that index. Else go to the next node.
3) Read the metadata from the node and get the stride length and number of entries. Extract bits to be matched from the destination address. Search for a match within the prefix entries using binary search. If an exact match is found, use the (=) next hop and assign it to current next hop. If the key falls in any range, use the (>) next hop pointer and return it. If the next node pointer is not null, repeat the same step, else return the current next hop.

Since that size of the data structure is designed to be equal or in multiples of cache line width, most parts of the node will be in cache after the first bytes are accessed. Also the locality in traffic patterns will keep most frequently used nodes in cache, so most lookups will be fast.

IV. Simulation Results and Discussion

A. Environment

The lookup operations were simulated on two Pentium based platforms running Windows 2000. Pentium II, 450 Mhz and Pentium 4, 2.4 Ghz system with 512 KB cache were used. The programs were written in C and compiled using 'Maximum Speed' optimization in Visual Studio 6. The code for Level Compressed tries was obtained from the authors and was compiled on the same platform with same settings.

B. Prefix Databases

The prefix databases used for simulations were obtained from [6]. The largest database was MaeWest with 93354 prefixes, while the smallest was MaeEast with 18360 prefixes. MaeWest database was very large in size on 27th April 2001, but its size reduced afterwards. It was used for simulations to study the behavior of Compact Prefix Tries for very large databases. While prefix databases are publicly available, it is not the case with traffic traces. The traffic was simulated assuming that every prefix has the same probability of being accessed. This assumption allows us to measure of the worst-case lookup time. In order to reduce the effects of cache locality, random permutations of all the entries in the forwarding table were used. The entries were extended to 32 bits by adding zeros.

C. Comparison with Other Schemes

1) Memory usage comparison: The memory requirements of Compact Prefix Tries are compared with various other schemes in the Table I. Since the databases used by other schemes were of different sizes, the comparison is not without flaws. The number of prefixes is not exactly same but, roughly 40000 prefixes in all the measurements.

Table I describes a comparison of the various schemes in terms of memory usage. The first 8 schemes in the table used the prefix size of approximately 40000 prefixes. The memory usage values reported in the literature are used for comparison. As can be seen from the table, Extended Compact Prefix Tries are second best in terms of memory requirements. The last row shows the memory required by Extended Compact Prefix Trie for MaeWest database with 93354 entries. Only 829 KB of space was required.

2) Lookup Performance Comparison: Ruiz-Sanchez *et. al* [12] have compared some schemes and there lookup times on a 200 MHz, Pentium-Pro based computer with 512 KB cache. They ran the simulations using MaeEast database with 47113 prefixes. As it was not possible to get the exact prefix database, a database of similar size from PacBell (48578 entries) was used for our simulations. The lookup performance was measured in same way as measured by Ruiz-Sanchez *et. al*. In our simulations, the time required for accessing the prefixes at each level of the Compact Prefix Tries was measured on Pentium II, 450 MHz system and scaled to 450 MHz clock. This comparison is also not without flaws because scaling up the clock dose not necessarily speed up lookup times by the same factor because memory access times do not speed up with faster clock.

Table II shows the lookup time variability for six different schemes. The lookup times for first five schemes are borrowed from [12]. Full expansion/compression scheme was the fastest scheme as it required just 3 memory accesses in the worst case. Extended Compact Prefix Trie performed much better than other trie compression schemes LC Trie and Multibit Trie.

3) Experimental comparison with LC-Tries: As the source code for LC-Tries was available, an experimental comparison was performed with them. Since the sources of other state of art software implementations [14] are not publicly available, a direct comparison with them could not made.

The lookup performance of LC Tries and Extended Compact Prefix Tries (ECP Tries) is compared in Table III. Results are shown only for Pentium 4, 2.4 GHz system. The fill factor for LC Tries was 0.5. ECP Tries are Compact Prefix Tries after using the extensions (initial array and large sized CSTNodes) discussed in Section III-D applied. The lookup performance was measured by randomly searching the prefixes in the database.

It can be seen from the table, that the Extended Compact Prefix Tries can perform lookup operations almost twice fast, and also require up to 33% less memory then $LC\text{-}Tries$. **Note:** For brevity we are unable to provide detailed results of our analysis of this new scheme. Such details can be found in [13].

Scheme	Entries	Size (KB)
Patricia Trie	38816	3262
6-way search on prefixes	38816	950
Binary search on hash tables	38816	1600
Full expansion/compression	43524	1057
Lulea scheme	32732	160
Controlled Prefix Expansion	38816	640
LC Trie	44168	708
Extended Compact Prefix Trie	44168	533
Extended Compact Prefix Trie	93354	829

TABLE I

MEMORY REQUIREMENT COMPARISON WITH OTHER SCHEMES

Scheme	10th percentile	50th (median) precentile	99th percentile
BSD Trie	2050	2640	3964
Full expansion/compression	115	213	373
Binary Search on pref. len.	484	702	3146
Multibit trie	364	591	1328
LC Trie	422	569	880
ECP Trie	177	422	635

TABLE II

PERCENTILES OF LOOKUP TIMES (ns)

Database	Prefixes	LC Tries		CP Tries		ECP Tries	
		Size	Mlps	Size	Mlps	Size	Mlps
Mae-east	18360	444	11.77	406	9.82	373	11.76
Aads	31283	585	5.41	541	8.02	456	10.00
PacBell	44168	708	4.22	606	6.73	533	8.83
Mae-west	93354	1259	3.13	1025	3.85	829	5.43

TABLE III

MEMORY AND LOOKUP PERFORMANCE COMPARISON WITH LC TRIES

(*Size is in KB and Mlps is million lookups per second)*

REFERENCES

[1] G. Cheung and S. McCanne, *Optimal Routing Table Design for IP Address Lookups Under Memory Constraints*, Infocom, 1999.

[2] P. Crescenzi and L. Dardini and R. Grossi. *IP Address Lookup Made Fast and Simple*, European Symposium on Algorithms, 1999, pp 65-76.

[3] M. Degermark, A. Brobnik, S. Carlsson and S. Pink, *Small Forwarding Tables for Fast Routing Lookups*, Proceedings of ACM SIGCOMM, October 1997.

[4] W. N. Eatherton, *Hardware-Based Internet Protocol Prefix Lookups*, Master's Thesis, Washington University, May 1999.

[5] E. Fredkin, *Trie Memory*, Communications of the ACM, 1960.

[6] IPMA. Routing Table Snapshots, http://www.merit.edu/ipma/.

[7] B. Lampson, V. Srinivasan and G. Varghese, *IP Lookups using Multiway and Multi-column search*, IEEE/ACM TON June 1999.

[8] Lmbench, *Tools for Performance Analysis*, http://www.bitmover.com/lmbench/.

[9] D. R. Morrison. *PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric*, J. ACM, vol 15, Oct. 1968, pp 514-34.

[10] S. Nilsson and G. Karlsson, *Fast Address Look-Up for Internet Routers*, Proceedings of IEEE Broadband Communications 98, April 1998.

[11] S. Nilsson and G. Karlsson, *IP-Address Lookup Using LC-Tries*, IEEE Journal on Selected Areas in Communications, June 1999.

[12] M. Ruiz-Sanchez, E. Biersack and W. Dabbous, *Survey and Taxonomy of IP Address Lookup Algorithms*, IEEE Network, March/April 2001.

[13] A. Sarda, *A memory-efficient scheme for IP Address Lookup using Compact Prefix Tries*, MS Thesis, Dept. of Computer Science and Engineering, Arizona State University, February 2003.

[14] V. Srinivasan and G. Varghese. *Faster IP Lookups using Controlled Prefix Expansion*, Proc. ACM Sigmetrics, 1998.