Trie Partitioning in Distributed PC Based Routers

Noel Athaide, Azeem Khan, D. Manjunath and A. Sahoo
IIT Bombay, Mumbai, India.
Email: athaiden,dmanju@ee.iitb.ac.in; azeem,sahoo@it.iitb.ac.in

***Abstract*— Recent research in PC based routers has proposed a distributed architecture. Such an architecture poses several challenges in the areas of scalability, robustness, efficiency of routing, latency and other issues. We examine the issue of decrease in throughput due to large routing tables in this architecture and propose partitioning as a solution. Our contribution is twofold: defining the concept of load for a node in a forwarding table trie and to show by simulation experiments the effectiveness of partitioning and its application to a distributed router.**

I. Introduction

Early routers were simple devices designed to support very limited functionality. The earliest routers were general purpose computers adapted to work as routers. Modern routers have significantly higher functionality. These include for example, supporting firewalls, offering encryption capabilities, QoS, per-user forwarding rules, very deep packet inspection and more. Changing requirements in packet processing capabilities demands flexibility in the router's architecture. This has led to successive generations of routers with increasing capabilities [1]–[6].

Open hardware and open source software based approaches offer advantages over closed hardware based designs [7]–[9]. A very interesting work used general purpose processors (DEC Alpha CPUs) to create a router [10]. The processors held instructions for the fast path execution code for IP header processing, in their on-chip memory. The forwarding tables were on two memory modules connected by a dedicated bus to the CPU. The execution code was extensively tuned so that the processor was pushed to its limit in processing incoming packet headers. IP packets not on the fast path were processed by another CPU which had a larger execution code in memory instead of on-chip. The general CPU also processed the control plane packets. This work showed that it is possible to create high performance routers using commodity CPUs and suitably tuned software.

Open source software offers diversity, quality and support for continuous improvement. Some software, e.g., [11], [12] have achieved considerable maturity. We believe that implementation of a new idea in software on a PC is easier and often quicker because there are many support tools available for development and troubleshooting. This motivates the use of commodity CPUs and open source software for building PC based routers.

The above ideas have been incorporated in recent works which propose the use of PCs as routers [13]–[15]. Bianco et al. [13] showed that using a Linux based system, they could achieve more than 90% of the injected throughput in the data plane. Using extensive experiments, Khan et al. [15] later showed that the performance of a single PC as a switching element was limited by its inherent architecture. Specifically, the limitation was in the I/O subsystem. The PCI bus on the PC, which is a shared bus design, limited the maximum bit rate. They also observed that this limitation could be overcome by using multiple PCs to act as a single logical router. Bianco et al. [14] also described a multistage architecture using several PCs that offered significant performance improvements over a single PC. Khan et al. [15] then experimented with a parallel processing based setup using PCs in a star topology and concluded that the multi-stage architecture proposed in [14] offered several advantages over the parallel based setup. Therefore we consider the multi-stage architecture in our work.

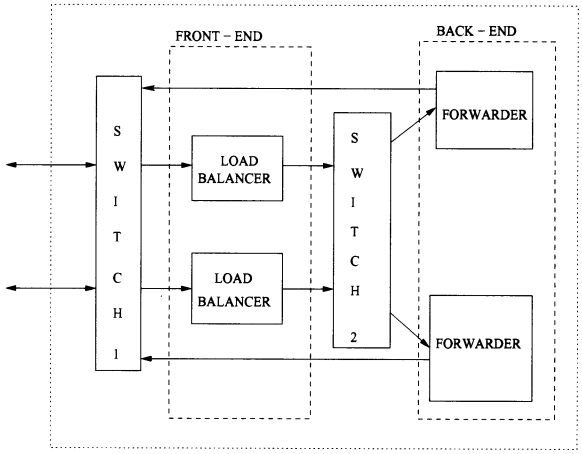

Fig. 1. Multi-stage Distributed Router. The outermost dotted box represents a single logical router to the outside world.

Fig. 1 shows the multi-stage PC architecture. Each of the machines in the front-end and the back-end is a PC. The front-end is connected to the back-end by a high speed switch—switch-2. The links terminating at the router terminate at switch-1. The direction of the arrows on the links show how the traffic flows through the router. The front-stage PCs direct the incoming packets to the back-stage. The front-stage may perform processing on the packets depending on its purpose. For example, if the front-end is a load balancer as shown in Fig. 1, it may use a simple scheduling mechanism like round robin to transmit the packets to the back-end. The back-stage PCs perform the layer-3 processing on the packets and act as forwarders of the packets. *All forwarders in the back-stage have identical forwarding tables.* After processing the IP header, the back-stage will transmit the packets to the next hop via switch-1

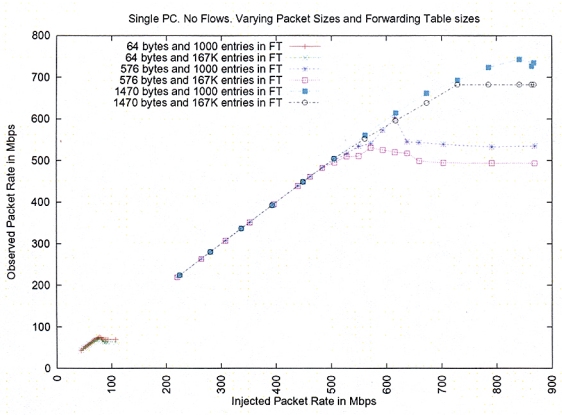

Fig. 2. Throughput Drop in a PC for large forwarding table

directly.

The distributed nature of the design introduces challenges in the data plane implementation like packet reordering [16], handling of fragmented packets, synchronization of the forwarding tables in the back-stage, different processor capabilities in the back-stage and intelligent load balancing in the back-end. The control protocol has to be designed to account for this distributed PC architecture. A control protocol design for this architecture has been proposed in [17]. The management plane also provides several interesting challenges in this architecture but it is outside the scope of this paper.

It was observed by Khan et al. in their work that the throughput of the individual PC acting as the forwarder in the back stage drops significantly as forwarding table size reaches approximately 100,000 entries [15] (Refer Fig. 2). The CPU and memory subsystem became a bottleneck for this table size. For small forwarding tables of 10,000 entries or less, the PC throughput was limited only by the I/O subsystem. Since, forwarding table sizes can only increase in the future, performance of the PC in the back-stage may worsen. Therefore investigate a method to have small forwarding tables on the back-stage PCs.

Small forwarding tables on the back-end PCs should not be achieved at the cost of losing routing information from the large forwarding table created by the router's routing protocol. Therefore, we may partition the large forwarding table into several smaller ones. For the multi-stage architecture, each of the partitioned smaller tables would reside on one back-stage PC. This leads us to issues such as, algorithms to be used, parameters used for partitioning, correctness of partitioning algorithm, application of the partitioning to the multi-stage architecture and synchronization of the partitions across back-end PCs. In this paper we address some of these issues. We have two main contributions in our work. First, for a forwarding table created using a trie, we define what is the load of a node and therefore the load of the trie. Our second contribution is to offer a method to reduce packet processing time for a distributed router design by applying the idea of partitioning to the IP address lookup process.

The remainder of the paper is organized as follows. In Section II we have an overview of data structures used in forwarding tables, some research proposals in partitioning forwarding tables and a preliminary definition of load which we will use for partitioning. Section III discusses the partitioning algorithm for a PATRICIA trie and simulation results for the partitioning. Section IV discusses the application of this algorithm to the PC based architecture.

The paper concludes in Section V, with a discussion on partitioning of other commonly used data structures and future work.

II. Preliminaries

Some of the tasks that take considerable amount of time in packet processing are IP header validation, forwarding table lookup, and fragmentation. Of these, the forwarding table lookup is the most expensive operation in terms of CPU usage [18]. To forward a packet, a router searches the forwarding table for the longest prefix match of the packet's destination IP address and transmits the packet through the outgoing interface of the matched entry. This process, commonly called *IP lookup*, involves (a) a search key; the destination IP of the packet, (b) entries in the forwarding table; *prefixes* and (c) the output interface as a result of the search; *next hop*. This lookup is non-trivial because there may be multiple matches and only the longest match should be used to determine the next hop [19]. Several efficient data-structures (for the forwarding table) and lookup algorithms have already been proposed in literature, to speed up the task of IP lookup, [6], [20]–[26].

In this work we seek to improve the performance of the distributed router architecture described in the previous section, by suitably partitioning the forwarding table to achieve faster IP lookups while efficiently using router resources. Partitioning of the forwarding table in routers has been explored in [27]–[31]. In [31], the authors separate the process of finding the egress line-card and the egress port on the line-card. Papers [27]–[29] discuss methods of partitioning the table so that a single IP lookup can be done in parallel, i.e., the same packet is simultaneously forwarded to several processors. However, our method is different in that we partition the forwarding table so that several processors may simultaneously process different packets. Moreover, we make sure that the partitions are load-balanced so that the processors share the IP lookup load in a balanced manner. The load of a partition is defined in Section II-B. The method used in [30] achieves the same objective with the difference that it only attempts to make small and equal sized partitions and does not balance the load between the partitions.

A. Related Work on Partitioning of Forwarding Table

In [27], the authors suggest partitioning the table with respect to output interfaces, i.e., all prefixes which share the same output interface are grouped into a single partition. Assuming that there is a one-to-one correspondence between a router's egress ports and next hop, it can be shown that the prefixes in each partition will be non-overlapping. As a result a search on

Lookup on T_1	Lookup on T_2	Result of XOR
0	0	0(Default gateway)
$l(u)$	0	$l(u)$
$l(x)$	$l(x) \oplus l(v)$	$l(v)$
0	$l(x) \oplus l(w)$	—

TABLE I

LOOKUP CASES WHEN SPLITTING THE TRIE AS IN FIG. 3

any partition will yield at most one match. So the problem of longest prefix matching is simplified to that of finding a single match. The partitions are searched in parallel using multiple processors, and the longest prefix from among them is selected. The main advantages of this method are:

- Parallel lookup architecture.
- Only one match per partition, and hence faster lookup on each partition and simpler data structure.
- No need to store output port number, since it is the same for all prefixes in a given partition.

On the other hand, [28] partitions the table according to the depth of the prefixes in a binary trie, i.e., all prefixes which are at the same depth in the binary trie are grouped into a single partition. Here too, the prefixes in a partition will be non-overlapping because in order for a prefix to overlap with another, it would (a) have to share a common path with that prefix and (b) be either above or below that prefix in the trie. Hence it cannot overlap and simulatneously have the same height as the other prefix. The search for the longest match is then carried out in a manner similar to that described in the previous method.

In [29], the prefixes are arranged in a binary trie which is then split into two as shown in Fig. 3. The next hops are expressed as binary numbers called *labels*, e.g., the next hop of prefix x is given by $l(x)$. The labels of the prefixes in the upper trie T_1 are left unchanged, while those of the prefixes in the lower trie T_2 are rewritten with the XOR of that label and the label of the nearest parent prefix in T_1. Now, both the partitions are searched in parallel. If a match is found, the corresponding label is returned else a 0 is returned. The results from both the partitions are then XOR-ed to give the final result. Table I shows the different possibilities. The last case is not possible, because if the lookup on T_2 returns a result then there will always be a parent prefix in T_1 which also match the given IP, hence the lookup on T_1 cannot return a 0.

This method of partitioning can be applied recursively to split the trie into as many partitions as required. The main advantage is that the XOR rule still applies, i.e., the results of the lookups on the individual partitions simply need to be XOR-ed, which is easily done in hardware, to get the correct next hop. The approach in [30] is different from the rest because it describes a method to partition the table for a multiple line card (LC) architecture, where each LC processes a different packet. Here prefixes which match at specific bit positions are grouped together. The bit positions used for such a partitioning, are decided by two criteria: (1) each partition should contain as few prefixes as possible and (2) the size difference between

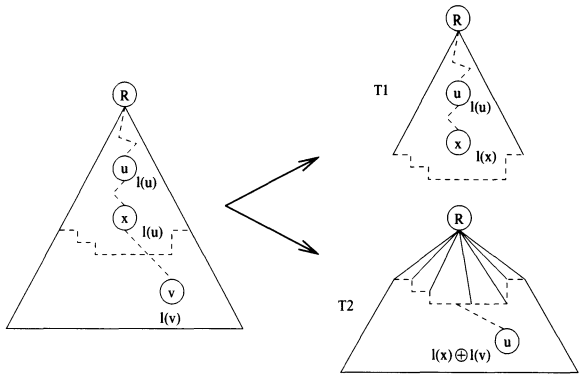

Fig. 3. Splitting the trie [29].

the largest and smallest partition should be minimum. However this method results in an increase in the number of prefixes because prefixes in which the specified bit position is a '*' or 'don't care' will occur in more than one partition. When a packet arrives at an LC these bit positions are inspected. If the LC does not contain the corresponding partition, then the packet is forwarded to the correct LC over the switching fabric. This partitioning method only attempts to make small and equal sized partitions and does not balance the load at LCs.

Since our focus is on increasing the throughput in the data plane, we describe in detail what kind of partitioning will help us achieve this objective. We defer control plane issues such as efficiently finding the best partition and dealing with routing updates to a later paper. In this work, we will look at partitioning of a PATRICIA trie [20], variations of which are used in routers today.

There is existing literature on graph partitioning for tree-like structures similar to PATRICIA tries [32]–[35]. However, as mentioned before, in this work our focus is not on developing an efficient algorithm for finding the partitions. Instead, we focus on describing what constitutes a load-balanced partition. Once this is defined, existing graph-partitioning algorithms can be suitably modified to actually find the partitions.

B. Load of a Trie

In this section, we formally define load of a trie in terms of IP lookup cost. When IP lookup in a router is performed using trie structures, the location of the nodes of the trie in memory/RAM could be arbitrary, and traversing the trie is done using pointers. IP lookup process starts at the root node of the trie and then continues down the trie incurring cost of an additional memory access for each node visited along the path. Thus, the load associated with an IP lookup is the cost incurred in memory accesses made to find the longest prefix match. Note that one may need to traverse the sub-trie below a node, before concluding that that node is indeed the longest prefix match. So, the number of memory accesses is not necessarily equal to the depth of the node corresponding to the longest prefix match. We do not include the computation cost of the processor in this definition of load, since memory is typically the bottleneck in

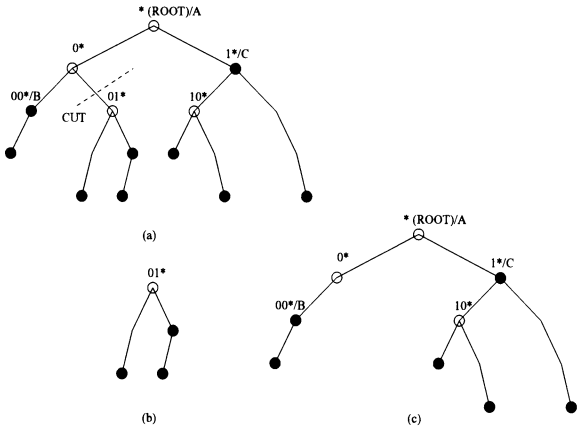

Fig. 4. Cutting a trie: (a) Original trie (b) Child sub-trie (c) Parent trie. Dark circles are used to indicate prefix nodes, while the light circles represent non-prefix (zero weight) nodes required for building the trie.

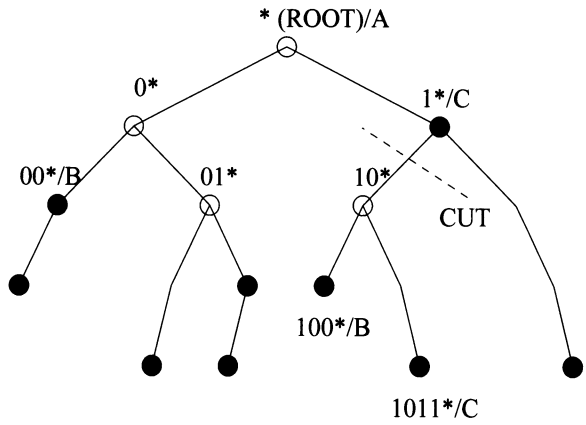

Fig. 5. Before partitioning, the string 10101100 will match the prefix 1* and the lookup will take three memory accesses (root-1*-10*). After partitioning, the lookup starts directly at the root of the child sub-trie (10*), and ends after one memory access. Thus the loads of parent and child tries after partitioning should be 0 and 1 respectively. However, simply adding the weights of nodes in each partition would give the loads of parent and child as 2 and 0 respectively.

such a scenario [36]. There are two types of nodes in a trie. A node having a next hop entry is referred to as a *prefix node*, whereas nodes having no next hop entries are called *non-prefix nodes*. We denote a trie as $T(V, E)$, where V is the set of nodes (both prefix and non-prefix) in the trie and E is the set of edges. We define the weight w_i of a prefix node i in the trie as the total load of all IP lookups for which node i is the longest prefix match, i.e.,

$$w_i = \sum_{k \in S} N_k \tag{1}$$

where S is the set of all IP lookups which have prefix i as the longest match, and N_k is the number of memory accesses for the k-th IP lookup from the set S. Since IP lookup can never result in non-prefix node as a longest prefix match, the weight of a non-prefix node is 0. The sum of the weights of all the nodes in the trie is then defined as the load of the trie.

$$W = \sum_{i \in V} w_i \tag{2}$$

where V is the set of nodes in the trie $T(V, E)$. Keeping this definition of load in mind, we shall proceed to the next section where we describe the partitioning of a PATRICIA trie.

III. PATRICIA's Breakup

We first consider the simpler case of partitioning the PATRICIA trie into two. We begin by assigning weights to each of the prefix nodes, as described in the previous section. The trie is then partitioned into two by (in a graph-theoretic sense) cutting it. A cut is identified by the edge e_{ij} between two nodes i and j. The cut splits the original trie into two tries: a *child* sub-trie rooted at node j, and a *parent* trie which is still rooted at the original root node and contains the node i as a leaf node. This is illustrated in Fig. 4. There are as many cuts possible as the number of edges in the trie. Our aim is to find the cut which will result in load-balanced partitions. We define a cut as a *balanced cut* if the difference between the loads of the two resulting tries is the least among all possible cuts. The optimization problem can be stated as:

$$e^* = \arg\min_{e \in E} \quad |W_p - W_c| \tag{3}$$

where e^* is the edge corresponding to the balanced cut, W_p and W_c are the loads of the parent and child tries respectively. Getting the balanced cut may not be as simple as it looks, because after the partition, the weights of the nodes in the new sub-tries change. Hence, while calculating the load of the child and parent sub-tries, the new weights of the nodes have to be considered. In fact, the computation of parent and child trie loads is non-trivial and we show the method of computing the new loads later in this section. To illustrate this point, we present Fig. 5 which shows how the parent and child load calculation can go wrong if we use equations (1) and (2) without modifying the weights. There are two reasons for this: (a) Lookups which start directly in the child trie after partitioning, go through fewer hops than in the original trie. Hence, the load of the child trie should decrease after partitioning (as compared to the load of this sub-trie in the original trie before partitioning). (b) Some IP lookups which have a longest match in the parent trie, may start directly in the root node of the child trie. This results in decrease in the load of the parent trie and increase in load of the child sub-trie. Thus, the formulae for load of parent and child tries must incorporate change in load due to reasons stated above. This is the reason why we are forced to do an exhaustive search over all possible edges. The algorithmic complexity of this algorithm is thus O(E). The frequency of operation of this algorithm is limited by the time taken to execute it on one side and the need for new partitions due to changes in the traffic pattern and/or changes in the routing information updates on the other side.

The nodes in the original trie need to maintain some more information to enable us to compute the correct load of the partitions. Each prefix node i stores two counters, a weight

counter (w_i) and a frequency counter (f_i). f_i is the number of IP lookups that have i as the longest match and w_i is the total number of memory accesses required by these IP lookups. Each non-prefix node j stores only a single counter m_j, which is the number of IP lookups for which the lookup does not continue below the node j. Let A_o, A_p and A_c be the set of prefix nodes in the original, parent and child tries respectively and let A_{NP} be the set of non-prefix nodes in the child trie that do not have any prefix node above them. Let $d(i)$ be the depth of node i in the original trie and d_c be the depth of the cut i.e., the depth of the root node of the child trie. Then the loads are given by:

$$W_o = \sum_{i \in A_o} w_i \tag{4}$$

$$W_p = \sum_{i \in A_p} w_i - \sum_{j \in A_{NP}} m_j.d(j) \tag{5}$$

$$W_c = \sum_{i \in A_c} (w_i - f_i.d_c) + \sum_{j \in A_{NP}} m_j.(d(j) - d_c) \tag{6}$$

Where W_o, the load of the original trie, is calculated in the same way as in equation (2). W_p and W_c are the load of the parent and child tries respectively after partitioning. The first term in (5) captures the IP lookups which terminate in the parent trie, while the second term captures the lookups which have the longest match in the parent partition but after partitioning, the lookup would start in the child trie. The first term in (6) accounts for all IP lookups in the child trie that have a longest prefix match in the child trie itself which will now have reduced depth because they would start out from the root node of the child trie. The second term accounts for the IP lookups which have the longest match in the parent partition but would start in the child trie after partitioning. These lookups would take fewer hops than in the original trie, the difference being equal the depth of the cut.

We can use the definition of a balanced cut to find N load-balanced partitions by cutting the trie in an iterative manner. In the first iteration it is divided into two partitions such that the loads are in the ratio $1 : (N-1)$. Then the sub-trie with load $(N-1)$ is partitioned into two such that the loads are in the ratio $1 : (N-2)$. This process continues until the last partition is obtained with loads in the ratio $1 : 1$. Thus, at the end of the iterative process, the original trie gets partitioned into N equal load tries.

Let us assume that in an iteration, the trie is partitioned into two tries such that the loads are in the ratio $1 : \alpha$. Then either the child or the parent can have a load of α. Accordingly, we define two cost functions which represent the load difference of the two partitions.

Let $W_p(T, e)$ and $W_c(T, e)$ be the loads of the parent and child tries respectively, after cutting trie T at edge e. We define two cost functions for calculating the weighted difference of loads of parent and child tries.

$$f_c(T, e, \alpha) = W_p(T, e) - \alpha.W_c(T, e)$$
$$f_p(T, e, \alpha) = \alpha.W_p(T, e) - W_c(T, e)$$

Obviously, we should choose the partition that has the minimum cost function. In Algorithm-1 we present a procedure for obtaining N partitions such that the partitions are as close to having equal loads as possible.

Algorithm 1 Procedure to find N balanced partitions

PROCEDURE $find\_N\_partitions(N, T)$ {N = number of partitions required; T = original trie}
for $k = 1$ to $N - 1$ **do**
  $\alpha = N - k$
  $e_1 = \arg\min_e f_c(T, e, \alpha)$
  $e_2 = \arg\min_e f_p(T, e, \alpha)$
  if $f_c(T, e_1, \alpha) < f_p(T, e_2, \alpha)$ **then** {Choose child sub-trie as k-th partition}
    $T_k = T_c$
    $T = T_p$
  else
    $T_k = T_p$
    $T = T_c$
  end if
end for

The iterative procedure in Algorithm-1 is illustrated in Fig. 6. Here, after 4 iterations, the original trie is partitioned into 5 parts, each with 1/5 the load of the original trie.

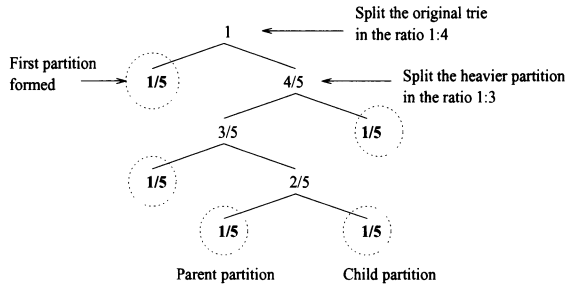

Fig. 6. The figure shown an example of creating 5 partitions. Note that the tree depicts the flow of the partitioning algorithm; it is not a trie. Each level of the tree structure represents a single iteration of partitioning. The root node indicates the load of the original trie normalized to 1. Each number in the tree represents the load of a trie relative to the load of the original trie. The left and right branches indicate the loads of the parent and child tries obtained by cutting a trie. Numbers in boldface represent the load of the final partitions.

The partitions thus formed are stored on different processing elements in the distributed router. When packets arrive at the router they need to be redirected to the appropriate partition to ensure that the lookup is done correctly. Therefore, before processing a packet, processing element must do a lookup of one more table, which we refer to as the *partition table*. The lookup of partition table involves matching the IP address against the root nodes of the partitioned sub-tries. Therefore, partition table lookup is quite simple and inexpensive. Organization of the partition table is illustrated in Table II.

- The first column contains the binary prefixes associated with the root nodes of each partition.
- The second column contains the partition-ID.

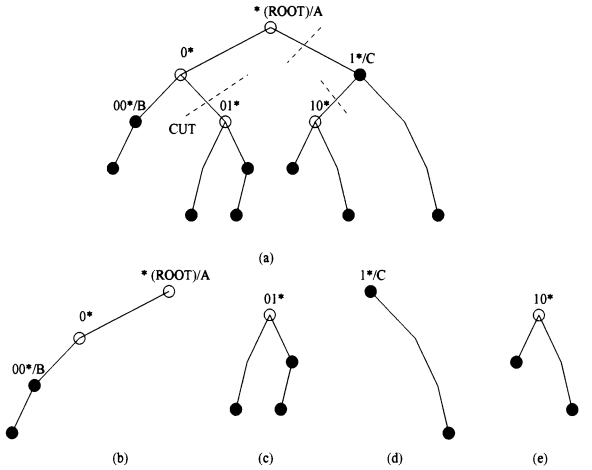

Fig. 7. (a) Original trie. (b), (c), (d), and (e) are partitions formed after cutting (a).

Partition rooted at	Partition number	Next hop
*	1	A
01*	2	A
1*	3	C
10*	4	C

TABLE II
PARTITION TABLE FOR FIG. 7

- The third column contains the next hop corresponding to the root node. If the root node of a partition is not a prefix node, the next hop of the nearest parent of the root node, in the original trie, is stored in this column.

Upon receiving a packet, the processing element does a longest prefix match search on the partition table, using the packet's destination IP, and forwards the packet to the correct partition. The next hop is temporarily stored and will be used if a better match is not found in the partition. The proofs for showing that IP lookups on a table partitioned in this manner returns the correct results are simple and listed in the Appendix .

At this point we must state certain implications/issues with this method of partitioning the forwarding table:

- We have assumed that the sample traffic trace used to assign weights to prefix nodes is representative of the traffic flow through a router. It is to be expected that should there be a short term traffic anomaly, where the traffic pattern deviates from expectation, the performance of our architecture may be worse than the round-robin load balancing scenario as in [14]. If this anomalous behaviour is common, then there is no such thing as a 'traffic pattern that is long enough to adapt to'. If that is the case, then there should be no partitioning.
- It is not possible to balance the load exactly, because of the granularity involved in cutting the trie.
- When a trie is cut, further path compression may be possible. Referring to Fig.4, we can see that after cutting, the parent trie can be compressed by removing node '0*'. Thus one would expect the actual average load at the parent trie to differ slightly from the load which was computed during partitioning.

A. Simulation Experiment

Partitioning of forwarding table makes sense in a parallel or distributed processing architecture. Therefore, in our simulation, we have assumed that there are two processors (i.e. lookup units LUs) performing IP lookups in parallel as shown in Fig. 8. Hence, we have two classifier elements (the load balancer or LB elements in Fig. 8) that take an incoming packet and make a decision on the processor that will do the lookup. Then the packet is handed over to that processor for actual IP lookup. The paths that a packet may take is shown by arrows in the illustration. For example, a packet arrives into the system at interface A. The LB element, based on some criteria, sends it to the second forwarder. The packet leaves from interface E and arrives at the second forwarder on interface F. After processing, it leaves the system from interface G. For our simulation experiments, we have considered two possible configurations.

In the first configuration, the forwarding table is unpartitioned and each forwarder contains the entire table. No classification is done by the load balancer elements. The packets are simply distributed between the processing elements (i.e. the two forwarders) in a round-robin fashion. This configuration is hereafter referred to as the *parallel processing* configuration. The time measured here is the average of t_{CD} and t_{RS}, where t_{CD} and t_{RS} are the IP lookup times up for the two processing elements (see Fig.8). The time taken by the load balancer to decide on the forwarder is negligible (since we used round robin) and is not a bottleneck and hence not taken into consideration for our measurements. Thus the time through the two forwarders gives us the time of processing packets in *parallel processing* configuration.

In the second confuguration, the forwarding table is partitioned into two. One partition resides in the first forwarder and the second partition resides in the second forwarder. The load balancer element performs a small table lookup to see which partition has the necessary information. It then sends the packet to the forwarder that contains the correct partition. For example, a packet arrives into the system at interface A. The load balancer element does a partition table lookup and decides to send the packet to the second forwarder. The packet leaves from interface E and arrives at the second forwarder on interface F. After processing, it leaves the system from interface G.

The time results for IP lookup using partitioning is the average of $(t_{AB}+t_{AE}+t_{CD}+t_{UV})$ and $(t_{PQ}+t_{PT}+t_{RS}+t_{FG})$, as shown in Fig.8, where $(t_{AB}+t_{AE})$ and $(t_{PQ}+t_{PT})$ are the partition table lookup times for each processing element, and $(t_{CD}+t_{UV})$ and $(t_{RS}+t_{FG})$ are the IP lookup times for each processing element.

Since we are doing a relative comparison of the two configurations, the time taken for packet transfers between elements are common and hence not considered. We are interested only in

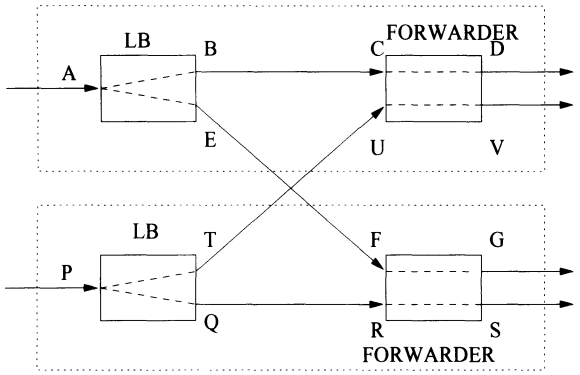

Fig. 8. **LB**: Load Balancer. The figure depicts two processing elements (dotted boxes) of a distributed router working in parallel. Each processing element stores only a part of the entire forwarding table, as determined by the partitioning algorithm.

	Original Table	**P1**	**P2**
No. of prefixes	41578 (1)	35959 (0.86)	5619 (0.14)
Load of Trie	1,481,729 (1)	681,690 (0.46)	719,235 (0.49)
Size of Trie	80448 (1)	69360 (0.86)	11088 (0.14)
nodes looked up	379,934 (1)	342,979 (0.9)	

TABLE III

PARTITIONING RESULTS FOR TRACE SET 1

reducing the computation time spent in each of the processing elements.

In our simulations, we have not measured the actual time taken for processing in each of the four elements. Instead, since the forwarding tables are stored as PATRICIA tries, we calculate the number of nodes looked up. This is done for two reasons. First, the time taken for a lookup varies greatly with a machine's design and hardware configuration. Second, our assumption has always been that each node lookup causes one memory access. Hence, counting the number of nodes looked up is a more accurate measure. For the partitioned table configuration, we have measured both the number of nodes looked up inside the partition table of the load balancers and the partitioned forwarding tables of the forwarders as described above.

We built the PATRICIA tries using three different trace sets. One is publicly available FUNET routing data set [37]. The other two sets were taken from the Lawrence Berkeley National Laboratory's Internet Traffic Archive [38]. The corresponding FUNET traffic trace [39] was also downloaded. For each set of traces and forwarding tables, we used the first half of the trace as a training set for computing node weights, while the second half was used for testing the effectiveness of the partitioning. After some trace analysis, we observed that more than 93% of the flows that existed in the first half of the trace continued into the second half of the trace. The loads of the partitions were computed for all possible cuts (exhaustive search) and the partitioning which gave the minimum difference in loads was selected as described in Section III.

For the FUNET data set, the simulation results are shown in Table III. The first three rows show the details of the partitions formed i.e. number of prefixes, weight and size for the tries formed out of this partitioning. Normalized values are shown in parentheses. The last row of Table III shows the time taken to process the same number of packets before and after partitioning. The number of nodes looked up in 'Original Table' indicates the average number taken by two processing elements working in parallel, with the complete forwarding table on both and without any partition table lookup. The number of nodes looked up in the partition section is the average of the two partitions' looked up nodes.

We can see that the load of the tries is well balanced, and that it does not necessarily translate to an equal balance in the number of prefixes in each partition or the number of nodes in the trie (size of the trie). We observe that there is a savings of about 10% in terms of memory accesses over the *parallel processing* configuration. Note that this is a relatively small forwarding table compared to the next two cases. Also, in this case, the training set was only 50,000 destination addresses.

	Original Table	**P1**	**P2**
No. of prefixes	100,000 (1)	56186 (0.56)	43814 (0.44)
Load of Trie	18,995,489 (1)	9,692,278 (0.51)	4,222,149 (0.22)
Size of Trie	185,114 (1)	103,382 (0.56)	81732 (0.44)
nodes looked up	1,670,305 (1)	1,454,892 (0.87)	

TABLE IV

PARTITIONING RESULTS FOR TRACE SET 2

	Original Table	**P1**	**P2**
No. of prefixes	165,463 (1)	119,835 (0.72)	45628 (0.28)
Load of Trie	5,502,546 (1)	2,842,691 (0.52)	1,187,093 (0.22)
Size of Trie	309,164(1)	224,004 (0.72)	85160 (0.28)
nodes looked up	1710637 (1)	1493441 (0.87)	

TABLE V

PARTITIONING RESULTS FOR TRACE SET 3

In the second trace whose results are detailed in Table IV, the forwarding table contains 100,000 routes. The training set used for the simulation was 600,000 packets with another 600,000 used for the simulation. In this case, the we observed that almost 82% of the flows in the first half of the trace continued into the second half of the trace. In this scenario, the routes were almost evenly distributed in the partitions unlike the earlier case. However, the loads are not evenly distributed at all. Despite this, the partitioning gives an improvement of about 13% over the plain *parallel processing* configuration.

In the third trace, we use a still larger forwarding table with 165,463 routes (See Table V). We observe that this time, the number of prefixes in each partition is heavily unbalanced unlike the earlier case. The loads too are quite unbalanced. Again, we observer that, despite the unbalanced partitions, there is improvement of about 13% over the *parallel processing* configuration.

We conclude from the observations of our experiments that partitioning the forwarding table in our setup results in savings in lookup time. The reduction in lookup time happens *despite*

the extra lookups that are done in the PT elements. This reduction in IP lookup time increases the throughput of our system. In our experiments, we observed a minimum of 10% and upto 13% improvement with partitioning over the *parallel processing* configuration.

We have analyzed partitioning of the PATRICIA trie structure for IP lookup for a multiple element configuration. Next, we examine partitioning in the context of a multistage distributed PC-based router.

IV. Application to Distributed PC Based Router

So far, the nature of partitioning has been independent of any particular hardware. We assumed the type of the data structure and the fact that memory accesses for non consecutive memory locations usually causes a new memory read. We now turn our attention to the PC hardware which was our motivation and attempt to apply our methods.

In Section I, we saw an approach advocated by [14], [15] which used a multi-stage distributed PC based router architecture. A slightly modified version of the proposal made in [14] appears in Fig. 1. There is a front-end of PCs that act as directors of incoming packets to the back-end. The actual layer-3 packet forwarding work i.e. IP header processing is done in the back-end. The front-end and the back-end are connected by a high speed switch—Switch-2 in the figure. The front-end can direct the packets based on some configurable criteria like load balancing at Layer-2. The back-end after processing the packets transmits them directly to the next hop via Switch-1.

Also, in Section I, we saw that large forwarding table sizes result in reduced throughput. Hence we use the approach in Section III to partition the forwarding table into smaller pieces. Each of the smaller pieces resides in one back-end PC. When a packet arrives at the incoming interface to the router, i.e. at the front stage, it must be directed to the correct partition. Each front-end PC must have a partition vector for the partitions. The front-end PC will examine the IP destination address of the packet and perform a partition lookup. The lookup tells the PC the location of the correct partition. The PC then directs the packet to the back-stage PC containing the correct partition. if the partitioned forwarding table size is small in the back-end PC, it should be limited only by its I/O subsystem. It should show higher throughput than in the case where it contains the unpartitioned forwarding table. Each back-end PC should benefit from the partitions and overall system throughput should increase.

There are two caveats here that are to be noted. First is, the cost of partition lookup in the first stage. If the partitioning method is very complex, the partition lookup process in the first stage may be computationally expensive. Should that happen, the first stage itself will become the bottleneck. It would then appear as if we have shifted the bottleneck of lookup time from the back-stage to the front-stage. Hence, a very complex partitioning should be avoided. A very complex partition lookup method must be avoided. The simple algorithm discussed in Section III gives good results. There is only a small additional complexity in the first-end PCs.

The second caveat happens if there are too many or too few PCs in the back-stage of our router. If the number of PCs in the back-stage are too small, few partitions are created. Then each partition may be still be large enough to cause a drop in throughput. It was seen in [15], the throughput drop is linear after a certain forwarding table size. Thus, despite its size, the large size partitions should still increase the throughput to some extent. In the second case, there are too many PCs in the back-stage. This will cause many small sized partitions to be created. This will increase the partition lookup time in the front stage. This problem can be solved using other means. For example, more than one PC may share the same partition and the partition lookup code in the first stage may choose any of these PCs. The choice may be made randomly or using round robin or some other method. The discussion of this scenario is outside the scope of this paper.

V. Discussions and Future Work

In this paper, we have proposed a method of partitioning a trie based forwarding table without loss of route information. At the same time, the loads across the partitioned tables are balanced relative to each other. We have also proposed a method to calculate the load of a trie based forwarding table. Smaller forwarding tables lead to overall reduction of per-packet processing time and hence an increase in throughput. Our simulation experiments show a clear reduction in the number of nodes looked up in the partitioned trie configuration. We have discussed how this method can be applied to a distributed router architecture.

There is a possibility of larger reductions in node lookups that was not observed by us. In our simulation experiments, the balanced cuts happened only at depths of 2, 1 and 1 for traces one, two and three respectively. Larger reductions are possible if the balanced cut occurs at a lower depth. In that case, the reduced number of nodes required to reach a heavy child subtrie would result in substantial savings in the nodes looked up.

PATRICIA tries were used in the early days of routing. But in the latest Linux distributions there are two approaches available. One is a Level Compressed (LC) trie, which is a variation of PATRICIA trie. The other one is a hash table based approach. LC trie is a method to reduce the number of memory accesses required for IP lookup. The idea is simple: given a binary trie, the i highest level is replaced by a single node of degree 2^i. The procedure is repeated on the sub-tries. Thus, in LC trie, each node may have multiple children. Our partitioning method should work directly on LC tries. However, Linux uses a space-efficient way of storing the LC trie [22]. In this compact data structure an IP lookup must traverse all the way to a leaf of the trie. This makes the load computation complex and hence our method cannot be applied directly.

In hash table based approach, Linux creates several hash tables for different prefix lengths in the routing entries. All routes with the same prefix lengths go into the same hash table. Hashing value of IP address is looked up in the hash table corresponding to the longest prefix length. If a match is found,

the search is terminated. Otherwise, the hash table of next smaller prefix length is looked up. The process continues until a match is found. In this approach, the prefixes are not stored in a trie-like structure. Hence our method cannot be applied directly. However, it may be possible to build a trie corresponding to the hash table and apply our partitioning method to this trie. But the load should now reflect the load of lookup in the original hash table, which may not be easy to do.

We are currently looking at the method of partitioning compact LC trie used by Linux. We are also looking into the effectiveness of multiple cuts i.e. multiple partitions and the tradeoffs thereof. We would like to compare the performance of both the vanilla LC trie and the space-efficient version used by Linux. We would also study how to apply our method to hashing based implementation of Linux. The main challenge in this case would be to compute load values corresponding to the hash table lookup.

VI. Acknowledgements

This work was supported by Ministry of Communications and Information Technology, Government of India under grant number 1(1)/2004-E-Infra.

Appendix

In the following, we show that the forwarding table partitioning described in algorithm 1 works correctly.

Lemma 1: The partitions created are disjoint.

Proof: Follows from the construction of the partitions. ■

Lemma 2: There are no duplicate entries in the partition table.

Proof: Follows from the construction of the partitions. ■

Lemma 3: The correct next hop is always returned.

Definition 1: Adjacent Partitions. Partitions P_1 and P_2 are said to be adjacent if $head(c) \in P_1$ and $tail(c) \in P_2$, where c is the cut which separates the partitions. We shall refer to P_1 as the *parent* partition and P_2 as the *child* partition.

Proof: We will use $i \prec j$ to denote that node i is a prefix of node j. Now consider a pair of adjacent partitions P_1 and P_2 where P_1 is the parent partition and P_2 is the child partition. We shall consider two cases here.

Case i: Suppose the longest matching prefix in the original trie corresponds to node k in the child partition. Let i and j be the root nodes of the parent and child partitions respectively. Then we will have

$$i \prec j \prec k \prec dest.IP$$

Nodes i and j will be listed in the partition table and lookup on this table will return node j as the longest match. Thus the packet will be forwarded to the LC containing the child partition. Since the child sub-trie is identical to the corresponding section in the original trie the node k will be returned as the longest match and hence the correct next hop is obtained.

Case ii: Suppose the longest matching prefix in the original trie corresponds to a node k in the parent partition. Again, let i and j be the root nodes of the parent and child partitions respectively. Then there are two possibilities:

(a)

$$i \prec k \prec \text{dest. IP}$$
$$k \not\prec j$$

Here a lookup on the partition table will result in the packet being forwarded to the LC holding the parent partition. Subsequent lookup on the parent partition will return k as the longest match and hence the correct next hop is returned.

(b)

$$i \prec k \prec j \prec \text{dest. IP}$$

Since k is the longest matching prefix in the original trie, j must be a non-prefix node. Also, any other node in the child partition which matches the destination IP must be a non-prefix node, else we would be contradicting the statement that k is the longest matching prefix. Now, a lookup on the partition table will result in the child partition being selected. However, since only non-prefix nodes match the destination IP, the lookup on the child partition will not return a result. Therefore, the next hop stored in the partition table will be used. This next hop is, by construction, the same as that of the nearest parent prefix p in the parent partition. Since $p \prec j$ we must have $i \prec k \prec p \prec j$. We conclude that $p = k$, else we would be contradicting the fact that k is the longest matching prefix. Thus the correct next hop is returned in this case too. ■

References

[1] V. P. Kumar, T. V. Lakshman, and D. Stiliadis, "Beyond best effort: Router architectures for the differentiated services of tomorrow's internet," *IEEE Communications Magazine*, vol. 36, no. 5, pp. 152–164, 1998.

[2] M. S. Blumenthal and D. D. Clark, "Rethinking the design of the internet: the end-to-end arguments vs. the brave new world," *ACM Trans. Inter. Tech.*, vol. 1, no. 1, pp. 70–109, 2001.

[3] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, 1997.

[4] A. Kumar, D. Manjunath, and J. Kuri, *Comunication Netwrking: An Analytical Approach*, Elsevier, 2004.

[5] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, Mar/Apr 2001.

[6] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *Proceedings of SIGCOMM '97*, September 1997, pp. 25–36.

[7] S. Karlin and L. Peterson, "Vera: an extensible router architecture," *Computer Networks*, vol. 38, no. 3, pp. 277–293, 2002.

[8] Y. Gottlieb and L. Peterson, "A Comparative Study of Extensible Routers," in *2002 IEEE Open Architectures and Network Programming Proceedings*, New York, NY USA, June 2002, pp. 51–62.

[9] S. Keshav, *An Engineering Approach to Computer Networking*, Pearson Education, 1997.

[10] C. Partridge et. al., "A 50-gb/s ip router," *IEEE/ACM Trans. Netw.*, vol. 6, no. 3, pp. 237–248, 1998.

[11] M. Handley, O. Hodson, and E. Kohler, "Xorp: An open platform for network research," in *Proc. of the 1st Workshop on Hot Topics on Networks*, 2002.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M..F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[13] A. Bianco, J.M. Finochietto, G. Galante, M. Mellia, and F. Neri, "Open-source pc-based software routers: A viable approach to high-performance packet switching," in *Proc. of the 3rd International Workshop on QoS in Multiservice IP Networks, QoSIP 2005*, 2005, pp. 353–366.

[14] A. Bianco, J.M. Finochietto, G. Galante, M. Mellia, D. Mazzucchi, and F. Neri, "Scalable layer-2/layer-3 multistage switching architectures for software routers," in *Globecom 2006*, 2006.

[15] A. Khan, R. Birke, D. Manjunath, A. Sahoo, and A. Bianco, "Distributed PC based routers: Bottleneck analysis and architecture proposal," in *Proceedings of High Performance Switching and Routing 2008*, May 2008.

[16] I. Keslassy and N. McKeown, "Maintaining packet order in two-stage switches," in *Proceedings of Infocomm 2002*, 2002, pp. 1032–1041.

[17] A. Bianco, R. Birke, J.M. Finochietto, and L. Giraudo et al., "Control and management plane in a multi-stage software router architecture," in *Proceedings of High Performance Switching and Routing 2008*, May 2008.

[18] Y. Luo, L. N. Bhuyan, and X. Chen, "Shared memory multiprocessor architectures for software ip routers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 12, pp. 1240–1249, 2003.

[19] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless inter-domain routing (CIDR): an address assignment and aggregation strategy," 1993.

[20] D. R. Morrison, "PATRICIA—practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968.

[21] S. Nilsson and G. Karlsson, "Fast address look-up for internet routers," in *BC '98: Proceedings of the IFIP TC6/WG6.2 Fourth International Conference on Broadband Communications*, London, UK, UK, 1998, pp. 11–22, Chapman & Hall, Ltd.

[22] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *Selected Areas in Communications, IEEE Journal on*, vol. 17, no. 6, pp. 1083–1092, Jun 1999.

[23] V. Srinivasan and G. Varghese, "Faster IP lookups using controlled prefix expansion," in *Measurement and Modeling of Computer Systems*, 1998, pp. 1–10.

[24] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *SIGCOMM*, 1997, pp. 3–14.

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

[26] J. Fu, O. Hagsand, and G. Karlsson, "Improving and analyzing lc-trie performance for ip-address lookup," *Journal of Networks*, vol. 2, no. 3, pp. 18–27, 2007.

[27] M.J. Akhbarizadeh and M. Nourani, "An ip packet forwarding technique based on partitioned lookup table," *Communications, 2002. ICC 2002. IEEE International Conference on*, vol. 4, pp. 2263–2267 vol.4, 2002.

[28] Z. Liang, K. Xu, and J. Wu, *A Scalable Parallel Lookup Framework Avoiding Longest Prefix Match*, Springer, Berlin / Heidelberg, 2004.

[29] G. Bongiovanni and P. Penna, "XOR-based schemes for fast parallel IP lookups," *Theor. Comp. Sys.*, vol. 38, no. 4, pp. 481–501, 2005.

[30] N. F. Tzeng, "Routing table partitioning for speedy packet lookups in scalable routers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 5, pp. 481–494, 2006.

[31] J. Fu, P. Sjodin, and G. Karlsson, "Two-stage ip-address lookup in distributed routers," in *Proceedings of the IEEE INFOCOM Computer Communications Wokshop*, Phoenix, AZ, USA, 2008, pp. 1–6.

[32] Y. Perl and S. R. Schach, "Max-min tree partitioning," *J. ACM*, vol. 28, no. 1, pp. 5–15, 1981.

[33] G. N. Frederickson, "Optimal algorithms for tree partitioning," in *SODA '91: Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, 1991, pp. 168–177, Society for Industrial and Applied Mathematics.

[34] T. N. Bui and B. R. Moon, "Genetic algorithm and graph partitioning," *Computers, IEEE Transactions on*, vol. 45, no. 7, pp. 841–855, Jul 1996.

[35] C.H.Q. Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and H.D. Simon, "A min-max cut algorithm for graph partitioning and data clustering," *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pp. 107–114, 2001.

[36] G. Varghese, *Network Algorithmics*, Elsevier, 2005.

[37] S. Nilsson, "A routing table for wordsized (32 bits) bitstrings implemented as a static level- and pathcompressed trie.," http://www.nada.kth.se/ snilsson/public/code/router/Data/funet.table.gz.

[38] Lawrence Berkeley National Laboratory, "The internet traffic archive," http://ita.ee.lbl.gov/.

[39] S. Nilsson, "A routing table for wordsized (32 bits) bitstrings implemented as a static level- and pathcompressed trie.," http://www.nada.kth.se/ snilsson/public/code/router/Data/funet.trace.gz.