

Efficient IP-Address Lookup with a Shared Forwarding Table for Multiple Virtual Routers

Jing Fu
KTH, Royal Institute of Technology
Stockholm, Sweden
jing@kth.se

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

ABSTRACT

Virtual routers are a promising way to provide network services such as customer-specific routing, policy-based routing, multi-topology routing, and network virtualization. However, the need to support a separate forwarding information base (FIB) for each virtual router leads to memory scaling challenges. In this paper, we present a small, shared data structure and a fast lookup algorithm that capitalize on the commonality of IP prefixes between each FIB. Experiments with real packet traces and routing tables show that our approach achieves much lower memory requirements and considerably faster lookup times. Our prototype implementation in the Click modular router, running both in user space and in the Linux kernel, demonstrates that our data structure and algorithm are an interesting solution for building scalable routers that support virtualization.

Keywords

IP-address lookup, virtual routers

1. INTRODUCTION

Virtual routers are a promising technology for supporting customer-specific routing, policy-based routing, multi-topology routing and network virtualization. The key challenges in virtual router technology are scalability and the performance, as a physical router needs to support tens or hundreds of virtual routers that each have a local forwarding information base (FIB). In addition, packets at high data rates arrive to the virtual routers interleaved, which further stresses the physical router as all FIBs are used simultaneously. To provide efficient packet forwarding and IP-address lookup, all FIBs should be preferably stored in low-latency memory. The size of the caches in general-purpose processors and ternary content addressable memories (TCAMs) in high-speed line cards limit the number of virtual routers that can be supported. For example, in Juniper's routers running logical router services, only up to 16 virtual routers are supported [1]. In this paper, we present an efficient data structure for storing multiple FIBs to reduce the memory usage. Our solution exploits the common prefixes (such as all Internet address blocks) across the virtual routers, while still

allowing each virtual router to have its own next-hops for each prefix.

1.1 The application of virtual routers

Customer-specific routing is one of the networking technologies that can rely on virtual routers. Large ISPs usually have a large number of customers and a variety of routes to other autonomous systems (ASes). The customers of an ISP may have varying requirements on routes. While customers in the financial industry and government agencies may prefer the most secure routes, customers that provide realtime multimedia services may prefer paths with low latency and high bandwidth. To satisfy customers' needs accordingly, ISPs should provide customer-specific routes depending on their requirements. However, in the current BGP decision process, each BGP router only selects a single 'best' route for each prefix, and announces it to the neighboring ASes (including customer ASes). To provide more flexible customer route selection, an ISP needs to have a router for each of its customers, which can be expensive if dedicated physical routers are used. Therefore, having multiple virtual routers on the same physical is an attractive alternative.

Policy-based routing [2] and multi-topology routing [3] [4] also lead to multiple FIBs on a physical router. In policy-based routing, instead of routing solely by the destination address, packets may also be routed based on the application, protocol or size of the packets. In multi-topology routing, there are independent topology configurations to provide service differentiation through class-based forwarding. A topology configuration is a subset of routers and links in a network. For example, for traffic that requires privacy, a subset of links and routers from the network that are secured can form a topology. While other unsecured links and routers are excluded from the topology. In these types of approaches, although it may not require virtual routers explicitly, however, several FIBs are required to forward packets.

In addition to the above mentioned networking technologies, several recent proposals in networking research may also rely on virtual routers. For example, virtual router technology can be used to simplify network management and reduce potential network disruptions [5] [6]. Also, in today's Internet, new network architectures and protocols can-

not be deployed or adequately evaluated, as the Internet is our communication infrastructure rather than researchers' experimental tool [7]. To address this problem, virtual networks are emerging as a way for researchers to evaluate new network protocols and architectures [8]. Network virtualization provides a way to run multiple virtual networks, customized for specific purposes, over a physical network, and it may be a long-term solution for the future Internet. In a virtual network infrastructure, a physical network may need to support up to hundreds of virtual networks, and subsequently, hundreds of virtual routers need to run on a physical router.

1.2 IP-address lookup in virtual routers

The key challenges in running virtual routers are scalability and the performance, as a physical router may need to support tens or hundreds of virtual routers. Therefore, scalability and high performance in packet forwarding and IP-address lookup is vital for the success of virtual routers. When performing an IP-address lookup, the router extracts the destination IP address from the packet, and determines the next-hop address by consulting the FIB. In current backbone routers, a FIB may contain more than 250k routing entries [9]. Each FIB entry contains a prefix (4 bytes), a prefix length (5 bits), and a pointer to the next-hop data structure (e.g., 11 bits to represent 2048 potential next-hops). Therefore, each entry is 6 bytes and all the entries together are about 1.5 MB. When using software lookup algorithms, the FIB entries need to be stored in a lookup data structure, which may require even more memory [10] [11].

When multiple virtual routers run IP forwarding concurrently, packets need first to be switched to the right virtual router. Thereafter, a separate data structure for each FIB can be used together with any of the current IP-address lookup algorithms. A disadvantage of having several FIBs is that a significant amount of memory is required to store them. For hardware-based approaches, larger TCAMs and SRAMs are needed to store the FIBs, which makes the system more complex and costly. When using software lookup algorithms on general-purpose processors, the lookup performance may decrease significantly due to the cache behavior. The cache of a general-purpose processor today is about 2 MB, and can be significantly smaller than the total size of all FIBs. Also, packets usually arrive interleaved to the virtual routers, which leads to that all virtual routers' FIBs are used simultaneously. Therefore, cache pollution will occur and virtual routers may swap out caches of one another, and results in decreased lookup performance.

In this paper, we present a shared data structure to store FIBs from all virtual routers. The goal of this data structure is not only to reduce its size, but also to enable efficient IP-address lookup in software routers. Our approach relies an approach called prefix transformation, which turns the network prefixes in all FIBs into the same prefix set. With the help of prefix transformation, the number of memory refer-

ences in each trie node is reduced from two to one during the lookup. Based on the shared data structure, we present our lookup algorithm. In addition, we present a prototype implementation of the algorithm in the Click modular router [12] supporting multiple virtual routers, both in user space and in the Linux kernel. Finally, we provide a thorough performance evaluation of the algorithm, analyzing the data structure size, packet lookup time, and packet forwarding speed.

The rest of the paper is organized as follows. Section 2 summarizes the background on IP-address lookup. Section 3 introduces the data structure to store the FIBs, and presents the lookup algorithm. The prototype implementation of the lookup algorithm in the Click modular router is shown in Section 4. In Section 5, we show the experimental setup for the performance evaluation. Section 6 presents and discusses the performance evaluation results. Finally, Section 7 concludes the paper.

2. BACKGROUND ON IP-ADDRESS LOOKUP

The current approaches for performing IP-address lookup use both hardware and software solutions. The hardware solutions make use of content addressable memory (CAM) and TCAM to perform IP-address lookup [13] [14] [15]. In software approaches, a modified version of a trie data structure is commonly used [10] [16] [17] [18] [19] [20] [21]. In this section, we first present an approach to optimize the FIB using prefix transformation, which is applicable to both hardware approaches using TCAMs and software approaches. Thereafter we present a trie-based data structure for software IP-address lookup.

2.1 Optimizing the FIB using prefix transformation

For both software and hardware lookup approaches, prefix transformation is a technique that can be used to reduce the FIB size and improve the lookup performance.

A FIB contains a set of prefixes, where each prefix has an associated length and next-hop address. Prefix transformation turns a set of prefixes in a FIB into a equivalent set; prefix sets are equivalent if they provide exactly the same lookup results for all possible IP addresses. There are a variety of ways to transform prefixes. One technique is called prefix compression, which aims at reducing the number of prefixes in a FIB by removing redundant entries [22]. For example, in a FIB with two prefixes 1^* and 11^* , if both prefixes have the same next-hop, then the prefix entry 11^* can be removed from the FIB. An IP address matching prefix 11^* will match 1^* instead and the packet will be forwarded to the same next-hop. A reduced prefix set requires fewer TCAM entries in hardware approaches, and leads to lower memory usage and better cache behavior in software approaches.

In addition to prefix compression, there is another transformation technique called prefix expansion, which expands the prefixes in order to provide more efficient lookups. In several approaches, prefix expansion is used to expand the

prefix	next-hop
P1 = 0*	N3
P2 = 000*	N1
P3 = 001*	N2
P4 = 010*	N1
P5 = 0100*	N4
P6 = 0101*	N3
P7 = 100*	N2
P8 = 101*	N3

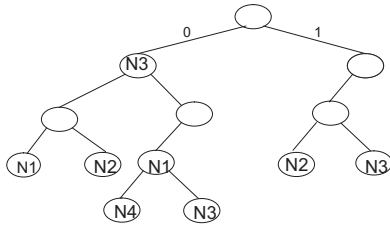


Figure 1: A FIB and its corresponding trie.

prefixes to a small number of distinct lengths [16] [17]. For example, a transformed prefix set may contain prefixes with lengths 8, 16, 24 and 32 only. A smaller number of distinct prefix lengths may be exploited by the lookup algorithm to improve the lookup performance.

2.2 Trie and level-compressed trie

Using the prefixes in a FIB as input, a trie can be constructed to store these prefixes for efficient lookup. A trie is a general data structure for storing strings [23]. In a trie, each string is represented by a node in the trie while the value of the string corresponds to the path from the root to the node. Prefixes in a routing table can be considered as binary strings, their values being stored implicitly in the trie nodes. Fig. 1 shows a FIB and its corresponding prefix trie. For each prefix in the FIB, there is a corresponding trie node. In the trie node, a pointer to the next-hop information which we name P is stored. In addition to the P , a trie node needs to contain pointers to its left and right child nodes. Note that the prefixes and their lengths are represented implicitly in the trie data structure, and therefore do not need to be stored in the nodes.

In a longest-prefix match, a given IP address is looked up by traversing the trie from the root node. Every bit in the address represents a path selection in the trie traversal: If the bit is zero, the left child is selected, otherwise, the right one is selected. During the trie node traversal, the P of the most-specific matching prefix so far is remembered. For example, when looking up an IP address starting with 011, at node corresponding to 0, N3 is obtained as a potential next-hop. Then the node corresponding to 01 is traversed, 01 does not contain a next-hop and there is no node corresponding to 011. Therefore, N3 is used as the next-hop since the longest-prefix match for the IP address is 0*. For an IP address starting with 0100, although the prefix 0* is a match in the first node, further trie traversal will determine that the prefix 0100* is the most specific match, and N4 will be used.

The original trie data structure is not very efficient, as the number of nodes for a whole routing table may be large and the trie may be deep. For IPv4 addresses, the search depth varies from 8 to 32 depending on the prefix length. Each trie node reference requires at least a memory access which may decrease the lookup performance. Therefore,

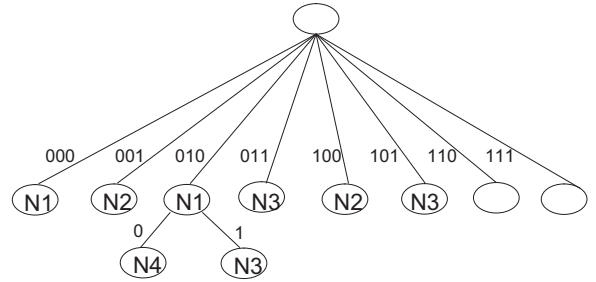


Figure 2: A level-compressed trie.

most of the software IP-address lookup algorithms use a modified trie data structure. For example, level compression is used in a number of lookup algorithms, where the i levels of a binary trie are replaced with a single node of degree 2^i [10] [16] [17]. The number of levels of a binary trie to be replaced can be denoted using a branch factor. The branch factor can either be static, or it can vary in different trie nodes. For example in LC-trie, the branch factor is computed dynamically, using a fill factor x between one and zero [10]. When computing the branching factor for a node k , the highest branching that produces at most $k(1-x)$ empty leaves is used. With a level-compressed trie, instead of only checking one bit at time in the lookup, several bits are checked each time to determine the corresponding child node. Fewer node traversals lead to fewer memory references which in turn results in faster lookups.

As an example, using level compression and fill factor $x = 0.5$, the trie in Fig. 1 can be transformed to a level-compressed trie shown in Fig. 2. The highest branching in the root node is 8, as it only results in 3 empty leaves, which is smaller than $8 \times (1 - 0.5) = 4$. If branching to 16 nodes, only 2 nodes in the fourth level are present and will result in 14 empty leaves, which is larger than $16 \times (1 - 0.5) = 8$. With a level-compressed trie, looking up an IP address starting with 000 only requires two node traversals, compared to the four node traversals in the original trie.

3. A SHARED FIB DATA STRUCTURE

When having multiple FIBs in a physical router, using a separate lookup data structure for each FIB leads to high memory usage, which may have scalability problems and result in poor lookup performance. In this section, we present a shared data structure to support efficient IP-address lookup. We first present limitations of a strawman approach to store the FIBs in a shared data structure. Thereafter we present our shared data structure and lookup algorithm.

3.1 Limitations of a strawman approach

When performing the lookup in a shared data structure, the input arguments are *virtual router ID* and *destination IP address*, and the return value is a *next-hop pointer P*.

A strawman approach is to store multiple FIBs directly on a single trie, as illustrated in Fig. 3. In the figure, there are

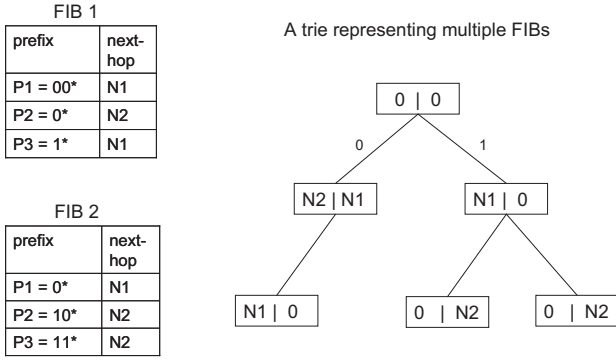


Figure 3: A prefix trie representing two FIBs.

two FIBs each containing 3 routing entries. In a trie node, instead of storing a single P , an array is used to store P in all FIBs. The number of elements in the array is equal to the number of FIBs. When a prefix associated with a trie node does not exist in a FIB, its P is denoted using a 0.

Using this trie data structure, each trie node needs to contain P for all virtual routers. If all P are stored directly in the trie node, then the node can be large. Assume that each P is one byte, 100 virtual routers may require 100 bytes. When traversing a node in the lookup, one memory access is not enough, as the *branch value*, *pointer to the child nodes* and the correct P cannot be obtained in a single memory access. An alternative is to only store a pointer to all P in the trie node. In this way, a trie node can be represented using 4 bytes only. However, this approach requires an additional memory access to obtain the P . Therefore, in both types of approaches, two memory references are required in each trie node traversal.

Note that in the traditional IP-address lookup with a single FIB, there is only one P in each node. Therefore, the branch value, pointer to the child nodes and P together can be encoded in 4 bytes, and one memory access is enough in a node traversal [10]. Two memory references per node traversal in the lookup occurs only when there are more than one FIB.

3.2 Generating a common prefix set

The strawman approach described above requires two memory references on each trie node due to the need to store all P , and therefore may significantly decrease the lookup performance. The goal of our approach is to support a single memory reference for each trie node traversal. The reason that P needs to be stored in all nodes is that any node can represent a prefix in the FIBs, and longest-prefix match is used in IP-address lookup. In a trie traversal, the algorithm needs to save the most-recent match. If there are no more-specific matches later in the traversal, the most-recent match will be used.

The solution is to turn the longest-prefix match into an exact match. By exact match we mean that an IP address

matches exactly one prefix in the FIB. In this way, all prefixes are stored in leaf nodes of the trie; since internal nodes do not contain prefixes, they do not need to store P .

The first step in constructing a shared data structure is to transform the set of prefixes in all FIBs into a common prefix set. In addition, the common prefix set should be both *disjoint* and *complete*: a prefix set is disjoint if the address ranges covered by all prefixes are all disjoint; a complete prefix set has the whole address range covered by the prefixes. With a disjoint and complete prefix set, there is an exact match for all possible IP addresses.

The procedure for constructing the common prefix set can be summarized as follows. First, all FIBs are represented in a trie as shown in Fig. 3. Then the prefix sets in the trie are turned into a common set that is both complete and disjoint. To make the common prefix set complete, a default prefix is added to all FIBs without a default route. The default drop prefix is of zero length and matches all IP addresses, and packets only match the default prefix are dropped. This makes the prefix sets complete since an IP address will match the default prefix if no other prefix matches. The trie in Fig. 4.a is complete after adding the default prefix (denoted $N0$) to the root node.

To make the prefix trie disjoint, *child pushing* can be performed recursively to replace all aggregated prefixes with leaves. The basic technique is to push the prefix in a node into its two child nodes. For example, prefix $P3 = 1*$ of FIB 1 in Fig. 3 represents all addresses that start with 1. Among these addresses, some will start with 10 and the rest will start with 11. Thus, the prefix $1*$ with length one is equivalent to the union of $10*$ and $11*$ with length two. In particular, the expanded prefix will inherit the next-hop of the original prefix. However, if there are two prefixes $1*$ and $10*$ with different next-hops, when expanding $1*$ to $10*$ and $11*$, the next-hop for $10*$ should be kept since it is more specific. The push operation is performed recursively starting from the root node. Fig. 4 shows the push operations that transform a prefix trie to a leaf-pushed common prefix trie.

After leaf-pushing to the trie shown in Fig. 4.c, both FIBs have a common prefix set with 4 entries: $00*$, $01*$, $10*$ and $11*$. The next-hops of these prefixes are also shown in the leaf nodes of the trie. The final step is to collapse the trie to reduce the number of entries in the common prefix set. If two leaf nodes of a subtree have the same P for all FIBs, then the two leaf nodes can be removed and we can set P to the subtree root node. In Fig. 4.c, it could be observed that $10*$ and $11*$ have the same P for both FIBs, therefore it can be collapsed to the prefix trie shown in Fig. 4.d, therefore removing one prefix in the common prefix set. The collapsing reduces the number of trie nodes and lowers the memory usage.

The time complexity to construct a disjoint and complete prefix set is $O(m)$, where m is the number of trie nodes. In the child-pushing phase, each node is visited once, starting from the root node. In the collapsing phase, a node can be

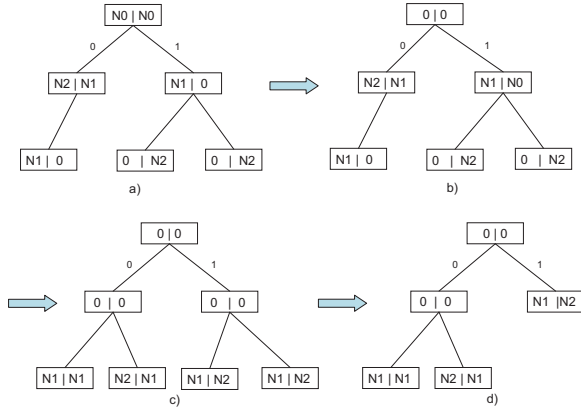


Figure 4: Transformation of a prefix trie to a common prefix set.

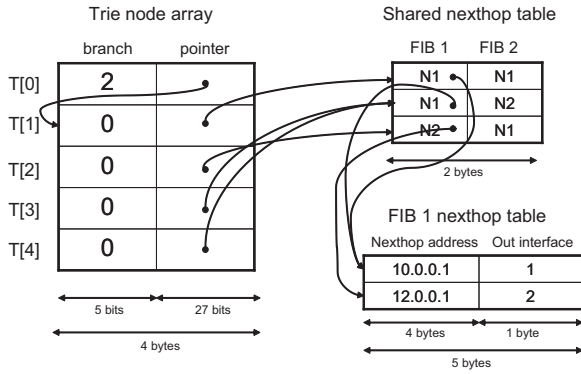


Figure 5: The lookup data structure.

visited once or twice. It starts from the root node, when reaching at the leaf nodes, the algorithm checks if it can be collapsed.

Note that the number of prefixes in the leaf-pushed common prefix set is larger than the number of entries in any leaf-pushed prefix set of an individual FIB. However, as the prefix sets are similar, the increase should be modest.

3.3 The shared data structure

When a common prefix set is obtained from the FIBs, a shared lookup data structure can be constructed, as shown in Fig. 5. The data structure contains a *trie node array*, a *shared next-hop table* and all FIBs' *next-hop tables*.

In the lookup data structure, the trie nodes are stored in an array T , and each element in the array is 4 bytes, to provide one memory access per trie node traversal. Each trie node element in the array contains a branch factor and a pointer. The branch factor represents the number of descendants of the node. As the number of descendants of a node is always a power of two, five bits are enough to represent all possible branches for IPv4 addresses. If the branch factor is zero, it means that the node is a leaf node. The trie root node is stored in $T[0]$, for an internal trie node, the pointer points to

the left most child. Since all children of a node are stored consecutively in the array, a pointer to the left-most child is enough – locations of other child nodes can then be derived. In the trie node array shown in Fig. 5, the root node $T[0]$ has a branch value of 2, which means it has $2^2 = 4$ child nodes, with the pointer pointing to the left-most child node $T[1]$. The child nodes correspond to prefixes 00^* , 01^* , 10^* and 11^* . All the child nodes are leaf nodes, and therefore their branch values are all 0. For a leaf node, the pointer points to an element in the shared next-hop table. As the number of trie nodes and the number of elements in the shared next-hop table both are significantly smaller than 2^{27} , therefore, each trie node only requires 4 bytes (5 bits for the branch factor, 27 bits for the pointer).

In the shared next-hop table, each element contains P for all virtual routers. We assume that the number of next-hops in a virtual router is less than 256, then one byte is enough for each P . If the number gets larger, 2 bytes can be used to point to 65536 potential next-hops. Also, 4 bits can be used if the number of next-hops is smaller than 16. The size of each element also depends on the number of virtual routers. In Fig. 5 there are two virtual routers, therefore each element is 2 bytes. The number of elements in the shared next-hop table is equal to or less than the number of leaf nodes in the trie, as the pointers in different nodes may point to the same element in the shared next-hop table. For example $T[3]$ and $T[4]$ in Fig. 5 both point to the second element in the shared next-hop table. In addition, the number of elements in the shared table is smaller than $\prod_{i=1}^n N_i$, where n is the number of virtual routers and N_i is the number of next-hops in virtual router i .

Constructing the data structure

The construction procedure of the shared lookup data structure is summarized as follows. First, all virtual routers' next-hop tables are constructed in a straightforward way. A next-hop table is an array where each element in the array contains the next-hop IP address and outgoing interface.

When all virtual router's next-hop tables are constructed, the next step is to construct the shared next-hop table. The shared next-hop table can be constructed by going through all prefixes in the common prefix set. For each prefix, an entry can be created if no similar entry exists. Going through all prefix entries and generate the shared next-hop table require $O(e)$, where e is the number of prefixes in the common prefix set. However, redundant shared next-hop entries need to be removed to reduce the memory usage. To remove redundant shared next-hop entries, they need to be sorted first. The fastest sorting algorithm such as *quick sort* is $O(e \log e)$ in time complexity, therefore the construction process for the shared next-hop table is $O(e \log e)$.

The last step is to construct the trie node array. One of the design choices is the branch value. The branch value can be fixed where it is the same in all nodes, thus, no bits are required in the trie node to represent the branch value. How-

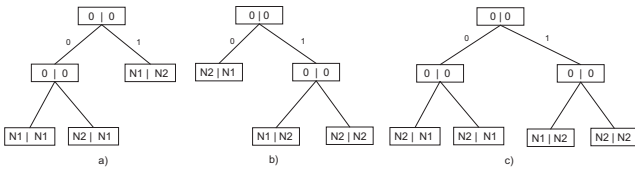


Figure 6: Trie update.

ever, we have used a technique from LC-trie to dynamically compute the branch factor in each node [10]. The branch value is computed using a fill factor of 0.5, where it requires 50% of the trie nodes to be present to allow level compression. Also, we used a fixed branching at the root node independent of the fill factor. The fixed branching to 2^{16} children gives the best performance, since most of the prefix lengths are longer than 16. To construct a trie node array from a disjoint and complete prefix set, the time complexity is $O(m)$, where m is the number of trie nodes.

Updating the data structure

The FIB lookup data structure may need to be updated, as prefixes can be added or removed from the FIB. When adding or removing a prefix, only the part of the trie corresponding to the prefix needs to be updated. Therefore, if the update is for a prefix with a length of /24 or /16, the update required on the prefix trie could be small, however, if it is the default route that changes, the whole prefix trie may need to be updated.

The procedure for updating the data structure when a prefix changes is first to leaf-push the modified part of the trie again. By comparing this modified part of the leaf-pushed trie before and after the update, the nodes that need to be added or deleted can be derived. For example, when updating a prefix, the previous leaf-pushed subtree is shown in Fig. 6.a, and the subtree after the update is shown in Fig. 6.b. In this case, the prefixes 01^* and 00^* can be removed, and the prefixes 10^* and 11^* need to be added in the trie.

Removing a prefix from the common prefix trie may result in large changes in the trie node array, as level-compression of the trie may also need to be changed. Therefore, prefix deletion is not performed in the leaf-pushed trie to reduce the update complexity. For example, instead of using the subtree in Fig. 6.b as the updated one, subtree shown in Fig. 6.c can be used. In this subtree, prefixes 00^* and 01^* are kept although they can be collapsed to a single prefix 0^* . However, the next-hops of the prefix 00^* is changed from $(N1|N1)$ to $(N2|N1)$. Also, the new prefixes 10^* and 11^* are added as shown in Fig. 6.c. To enable incremental updates and allow new trie nodes to be added in the end of the array, extra memory should be allocated when constructing the initial trie node array.

When the new trie nodes are added incrementally, the pointer to the shared next-hop entry needs to be set. If the shared next-hop entry does not already exist in the table, the entry needs to be created in the end of the table, therefore, more memory also needs to be allocated in the shared next-hop ta-

```

1  byte lookup(int vn_id, int ip_adr){
2      node = T[0];
3      pos = 0;
4      branch = node.branch;
5      ptr = node.pointer;
6      while (branch != 0) {
7          node = T[ptr + EXTRACT(pos, branch, ip_adr)];
8          pos = pos + branch;
9          branch = node.branch;
10         ptr = node.pointer;
11     }
12     return shared_nextHop[ptr][vn_id];
13 }

```

Figure 7: The lookup code with a shared data structure.

ble when it is constructed. Also, when constructing a FIB's next-hop table, more memory can be allocated to allow incremental updates.

Note that the incrementally updated data structure may not be the same as if the data structure is created from the scratch. Therefore, it may not be memory and lookup efficient after a large number of updates. An approach to deal with this is from time to time the router reconstructs the lookup data structure from the scratch.

3.4 Lookup algorithm

When the shared data structure is designed, the lookup algorithm can be implemented very efficiently as shown in Fig. 7. The lookup function takes the *virtual router ID* and the *destination IP address* as arguments, and returns the correct P .

The lookup starts at the root node $T[0]$, which is the first element in the trie node array. In line 3, the initial position of the IP-address string pos is set to 0. In lines 4-5, the node's branch value $branch$ and pointer to the left-most child node ptr are obtained. Line 6 checks if the node is a leaf node, and in general the root node is not a leaf, and the program continues to line 7.

Line 7 tries to find the correct child node matching the search string ip_adr . $EXTRACT(pos, branch, ip_adr)$ is a function that returns the number given by the $branch$ bits starting from position pos in the binary string ip_adr . Basically, the EXTRACT function returns the position of the correct child node relative to the left-most child node. By adding the return value from the EXTRACT function with the pointer to the left-most child node ptr , the correct child node is obtained.

Reaching at the correct child node, the pos is updated in lines 8, and the child node's $branch$ and ptr are obtained in lines 9-10. In the next iteration of the while loop in line 6, it is first checked if the node is a leaf node. If it is not a leaf node, the loop continues. Otherwise by following ptr from the leaf node in line 12, the element in the shared next-hop table matching the prefix is obtained. Depending on the

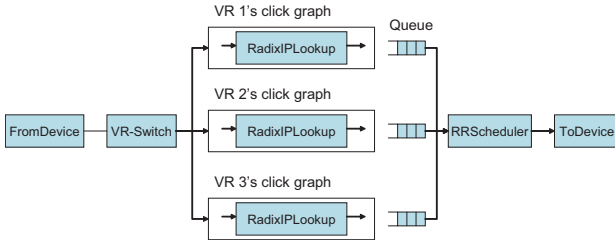


Figure 8: A Click configuration supporting multiple virtual routers.

virtual router ID, the correct P is returned.

4. PROTOTYPE IMPLEMENTATION IN CLICK ROUTER

As a proof of concept, we implemented a prototype of the shared data structure and the lookup algorithm in the Click modular router [12].

4.1 Click

The Click modular router is an architecture for building flexible software routers. A Click router is assembled from packet processing modules called elements. Each element implements packet processing functions like classification, queuing, scheduling and interface with the network device. A router configuration is built by connecting the elements into a directed graph; with packets flow along the graph's edges.

A large number of elements have been developed in Click as a part of the standard Click distribution, where each element can be customized to run in user space and/or in the Linux kernel. The Click router is extensible as new elements can be developed to provide customized processing tasks.

4.2 A Click configuration supporting multiple virtual routers

An approach to support multiple virtual routers in Click is to construct a single Click configuration for all virtual routers, as shown in Fig. 8 [24]. In the figure, a *FromDevice* element polls the packets from the Ethernet interfaces. After that, the packets are switched by a *VR-Switch* element to different virtual routers. In each router's Click configuration, packets are processed, and destination IP addresses are looked up using a *RadixIPLookup* element. Note that a virtual router's Click graph may contain more than a single element, and Click graphs in different virtual routers may vary. After being processed by the virtual router, the packets are put in a *Queue*, and scheduled using a *RoundRobinScheduler* element. Finally, the packets are sent out to the link using a *ToDevice* element.

Using this Click configuration, multiple virtual routers are supported. To allow CPU and bandwidth sharing, the virtual routers' Click graph can be executed by different threads, each being active for a period of time.

4.3 A Click element supporting a shared data structure

A limitation of the Click configuration in Fig. 8 is that an instance of the *RadixIPLookup* element is used in each virtual router. In each instance, a separate trie data structure containing the virtual router's FIB needs to be stored, which leads to high memory usage. Also, each virtual router thread will only be active for a while, and during this active period, the thread needs to swap out and swap in data structure from the cache, which results in poor cache behavior.

To deal with the problem, we have implemented a Click element *CombinedIPLookup* that runs both in user space and in the Linux kernel. In each virtual router, an instance of the *CombinedIPLookup* element is created. However, all instances of the element have a pointer to a common data structure, *CombinedADT*. The *CombinedADT* data structure contains the *trie node array* and the *shared next-hop table*, while each virtual router's *next-hop table* is stored in the *CombinedIPLookup* element. The lookup in all virtual routers are performed on this single instance of the data structure, which leads to smaller memory usage, better cache behavior, and improved lookup performance.

Similar to all other IP-address lookup elements in Click, the *CombinedIPLookup* element supports the four basic API functions. The first function is *configure*, which reads in the whole routing table from a stream buffer. The second function is *lookup*, that determines the next-hop address of a packet. The last two functions are *add_route* and *delete_route*, which dynamically add and delete prefixes from the FIB.

In a Click element for IP-address lookup, the initialization process of constructing the data structure is performed in the end of the *configure* function, after reading in the whole routing table. In *CombinedIPLookup*, the initialization process is modified slightly, as the process of constructing the data structure is not called by the *configure* function directly. When all Click elements' *configure* function have been executed, the initialization process constructs the shared data structure.

5. EXPERIMENTAL SETUP

In this section we describe the experimental setup for the performance evaluation of the lookup algorithms. Routing tables and packet traces are not easy to obtain due to privacy reasons. Still, we have done our best to obtain the routing tables from several sources, and a real packet trace, to correctly evaluate the performance.

5.1 Routing tables

To evaluate the performance of lookup with multiple FIBs, a number of routing tables are needed. In general, the lookup benefit of a shared FIB will be higher if the prefixes in the FIBs are more similar. To not over-estimate the performance of a shared data structure for lookup, we have downloaded routing tables from several sources, so that the FIBs' prefixes are not too similar to one another.

The first set of routing tables are two tables from the Finnish University Network (FUNET). The routing tables were downloaded at CSC’s FUNET looking glass page in May 2005 [25]. The second set of tables are from the Swedish University Network (SUNET), which consists of 54 routers [26]. We downloaded the table using Cisco command *show ip route* in the routers’ command line interface in October 2006. Also, we downloaded seven tables from the University of Oregon’s Route Views project in May 2008 [9]. Finally, the size of the routing tables also varies. Most of the FUNET and SUNET routing tables contain 160k - 180k entries, while some of the tables in Route Views project contain more than 250k entries.

When performing the experiments with virtual routers, each virtual router uses one of the routing tables. In the case that there are less number of virtual routers than available routing tables, we randomly select routing tables to use. To provide more accurate results, we perform a number of simulations using different routing tables, and compute the average values.

5.2 Packet traces

To study the packet lookup performance, packet traces containing destination IP addresses are required. We used two sets of traces for our evaluation.

The first set of traces have been synthetically generated by choosing prefixes randomly from a uniform distribution over the routing entries in the routing tables. In these traces, every entry in the routing table has an equal probability to be chosen. For each virtual router, a trie is generated based on its routing table. The length of all traces together is approximately 20 million packets.

The second set of traces is based on a real packet trace containing more than 20 million packets with destination addresses. We break the original trace into disjoint parts and use them as traces to different virtual routers. The trace is captured on one of the interfaces of FUNET’s Helsinki core router in March 2005. The router is located at Helsinki University of Technology and carries the FUNET international traffic. The interface where the trace was recorded is connected to the FICIX2 exchange point, which is a part of Finnish Communication and Internet Exchange (FICIX). It peers with other operators in Finland. In addition to universities in Finland, some of the student dormitories are also connected through FUNET. The trace itself is confidential due to the sensitive nature of packet. We were unable to find any trace publicly available on the Internet containing real destination IP addresses.

We assume CPU and bandwidth sharing among the virtual routers, therefore the arrival of packets to different virtual routers are based on round robin. Basically, the first packet is for virtual router 1, and the second packet is for virtual router 2, and so on.

5.3 Experimental platform

We used a computer from the Emulab facility to perform the experimental studies [27]. We used a pc3000 system in Emulab with the following specifications: 3.0 GHz 64-bit Intel Xeon processor with a 8 kB L1 and 2 MB L2 data cache. The main memory is a 2 GB 400 Mhz DDR2 RAM.

6. PERFORMANCE EVALUATION

In this section, we compare the performance of having a shared data structure with a separate data structure. In particular, we analyze the data structure size and packet lookup time.

6.1 Data structure size

We first analyze the size of the data structures, for both approaches with a shared FIB and a separate FIB.

We let S denote the total data structure size with a shared FIB containing n FIBs, which is also the number of virtual routers. As can be seen in Fig. 5, S is the sum of the size for the trie nodes, shared next-hop table, and all FIBs’ next-hop tables, therefore $S = S_t + S_s + \sum_{i=1}^n S_f^i$, where S_t is the size for the *trie node array*, S_s is the size for the *shared next-hop table*, and S_f^i is the size of FIB i ’s *next-hop table*.

We further let S' denote the sum of data structure sizes with separated FIBs. As a shared next-hop table is not required, each FIB data structure needs to contain its own trie node array and next-hop table. The total size is as follows: $S' = \sum_{i=1}^n (S_t^i + S_f^i)$, where S_t^i is the size of FIB i ’s trie node array and S_f^i is the size of FIB i ’s next-hop table.

We now analyze in detail the components that contribute to the total data structure sizes in both cases.

The size of the trie node array

The size of the trie node array S_t^i depends solely on the number of entries in the trie node array, as the size of each element is 4 bytes. The number of elements in the trie node array is loosely related to the number of FIB entries, though it is not exactly the same due to leaf-pushing, collapsing of the trie and construction of the level-compressed trie. For example in the FIB shown in Fig. 1, there are eight FIB entries. However, its level-compressed trie shown in Fig. 2 has 11 nodes in total. In this example, the number of trie nodes is 38% higher than the number of FIB entries. Further experimental results show that the number of trie node entries is approximately 1.5 times the number of FIB entries. Therefore in a FIB with 250k entries, the total size of the trie node array is approximately $2.5 \times 10^5 \times 1.5 \times 4/10^6 = 1.5$ MB.

When using a shared FIB, the number of entries in the trie node array is in general larger than entries in a single FIB. For example, there are two FIBs A and B, in FIB A’s data structure, there are 300k entries in the trie node array while there are 400k entries in the trie node array of FIB B. In the shared FIB data structure, there could be for example 450k entries. We let R denote the *increase ratio* for the size of the trie node array as follows: $R = n \times E_s / \sum_{i=1}^n (E_i)$. E_s is the number of trie node entries in the shared FIB, E_i is the

Table 1: The average increase ratio for the size of the trie node array.

Number of FIBs	2	5	10	20	50
Increase ratio	1.1	2.4	2.7	2.8	2.9

number of trie node entries in FIB i , and n is the number of virtual routers or FIBs. Using this formula, the increase ratio R for the example above is $(2 \times 4.5 \times 10^5) / (3.0 + 4.0) \times 10^5 = 1.29$.

Table 1 shows the average increase ratio R for the trie node array with different numbers of FIBs. This number is highly based on the FIBs used to perform the study. For example, when using FIBs from the same source such as the SUNET tables, the increase ratio could be small, below 1.1. While using FIBs obtained from different sources, for example one from the Route views project and the other from FUNET, the increase ratio could be larger. We have performed a large number of experiments to get an average.

Using the increase ratio, the size of the trie node array for a shared FIB can be computed. For example, if the average array size for a FIB is 1.5 MB, and then the array size for a shared FIB containing 50 FIBs is $1.5 \times 2.9 = 4.4$ MB, where 2.9 is the experimental value from Table 1.

The size of the shared next-hop table

Using the shared data structure approach, an extra shared next-hop table is needed. The size of an entry in the shared next-hop table depends on the number of FIBs, and it is 1 byte for each FIB. The number of entries is always smaller than the number of leaf nodes in trie, and also smaller than $\prod_{i=1}^n N_i$, where n is the number of virtual routers and N_i is the number of next-hops in virtual router i .

However, both the number of leaf nodes and $\prod_{i=1}^n N_i$ can be large. The number of leaf nodes in a trie can be around 200k entries, and $\prod_{i=1}^n N_i$ can be $5^{10} = 9.8 \times 10^6$ if there are 10 FIBs where each FIB has 5 next-hops.

We believe that the number of shared next-hop entries can be significantly smaller. Therefore, we have performed an experimental study, with results shown in Table 2. The table shows for example that for a shared FIB data structure containing two FIBs, there are only 18 shared next-hop entries. Note that 18 is a average number based on a number of simulations, and it can vary in different simulations. In the most common case, each FIB has 4 next-hops, therefore there are at most 16 entries according to $\prod_{i=1}^n N_i$. However, when one of the FIBs has 40 next-hops, the number of shared next-hop entries becomes larger.

Although the number of entries may vary for a small number of FIBs, however, when the number of FIBs increases, the number of entries stabilizes. For example, when using 50 FIBs, the number of shared next-hop entries is close to 1900 in most of the simulations. As this number is signif-

Table 2: The average number of entries in the shared next-hop table.

Number of FIBs	2	5	10	20	50
Number of entries	18	170	1600	1700	1900

Table 3: The total data structure size (MB).

Number of FIBs	2	5	10	20	50
A shared FIB	1.6	3.6	4.1	4.2	4.4
Separated FIBs	3.0	7.5	15	30	75

icantly lower than 2.0×10^5 , one of the upper bounds we earlier identified, it is very interesting. It basically suggests that we need no more than 2000 entries in the shared next-hop table when there are 50 FIBs, and the total data structure size is only about 100 kB, which is very small relative to the size for the trie node array.

The size of FIB specific next-hop tables

The number of entries in an individual FIB's next-hop table varies between 3 and 40 among a variety of routing tables we studied. Each element in the table is only 5 bytes, so the total table size can be at most 200 bytes. As this number is significantly smaller than the size of the trie node array, it can be neglected in the study.

Total data structure size

Based on the results from previous subsections, we compute the total data structure sizes for both approaches, with the results shown in Table 3. The results in the table show that a significantly smaller data structure is required if a shared FIB is used. For example, when there are 50 FIBs, a shared data structure only requires 4.4 MB, while the total size for all separated FIBs is about 75 MB.

6.2 Packet lookup time

We have shown in the previous subsection that a shared FIB data structure significantly lowers the memory requirements. In hardware technologies based on TCAMs and SRAMs, this means reduced memory requirements, system complexity and cost. When using software lookup algorithms on general-purpose processors, the packet lookup time is one of the key performance metrics. Our goal is that the reduced memory usage leads to better cache behavior and faster lookup. The packet lookup time can be divided into two components: instruction execution time and data access time.

Instruction execution time

The instruction execution time is the product of the number of executed instructions and the time required to execute a single instruction.

The number of executed instructions for a lookup depends on the trie search depth. For each trie node traversal, the code lines 6-11 in Fig. 7 are executed. Based on our experimental study, it shows that the average trie search depth is between one and two, therefore, there are about two or three node traversals in each lookup. Based on the compiled assembler code, 60 instructions are executed in average for each lookup.

The time required to execute a single instruction depends on the clock frequency and the number of instructions executed per clock cycle. Our test computer has a clock frequency of 3.0 Ghz, and we assume the computer performs 2 instructions per clock cycle, loosely based on the SPEC measurement results [28]. Then the time for execute a single instruction is in average 0.10 ns, although it may differ depending on the instruction. Therefore, the total instruction execution time of a lookup is about 6 ns. Note that the instruction execution time with a shared FIB is similar to the approach of using separated FIBs, as the number of instructions is very similar.

Data access time

Modern processors generally have L1 cache, L2 cache and the main memory. A L1 cache access requires less than 3 ns, and a L2 cache access could require about 8 ns. When an item is not found in cache, the main memory is accessed, and a main-memory access requires about 60 ns. So, depending on whether a data item can be found in the cache or not, the access time varies significantly. Also, as the instruction execution time is only about 6 ns, therefore a significant part of the packet lookup time is spent on data accesses when cache misses occur.

The caches of a general-purpose processor today are about 2 MB, therefore, with a shared data structure, most memory access might be found in the cache. While using separated FIB data structures with a large number of FIBs, cache misses may occur more frequently, which may result in significantly increased packet lookup time.

The data access time may vary significantly due to the arrival process of the packets (i.e., packet traces). When an IP address is looked up, the trie nodes are traversed and will be brought into the cache. Next time the same IP address (or another IP address belonging to the same network prefix) is looked up again, the trie nodes can be found in cache and accessed faster. So if a packet trace contains many packets with the same destination address, the data access time will be shorter.

Total packet lookup time

We have performed an experimental study of the packet lookup time using the test computer. We studied the average lookup

Table 4: The average packet lookup time (ns).

Number of FIBs	2	5	10	20	50
FUNET trace, a shared FIB	8.8	10	11	11	11
FUNET trace, separated FIBs	14	20	22	34	64
Random network trace, a shared FIB	48	88	103	105	106
Random network trace, separated FIBs	90	120	150	200	230

time with a random network trace and the FUNET trace. The average lookup time is obtained by dividing the total lookup time with the number lookups. In an experimental run, more than 20 million packets are looked up to measure the correct average. The lookup time for a single packet, however, depends on the trie search depth and cache behavior, and is difficult to measure directly as it is in nanosecond range.

Table 4 shows the average packet lookup time result with a shared FIB and with separated FIBs. When we compare the result of a shared FIB with separated FIBs, the figure shows that a shared FIB leads to significantly faster lookup. For example, when there are two FIBs, the shared data structure nearly doubles the lookup performance. When there are 50 FIBs in a shared data structure, the speedup could be up to five times better.

Another observation from the table is that the FUNET trace leads to significantly lower packet lookup time compared to the random network trace. The lookup performance for the FUNET trace is about 4 to 9 times better as shown in the table. The faster lookup time due to that the FUNET trace has a smaller subset of prefixes than the random network trace. Also, some prefixes are more popular than others in the FUNET trace, and lookup of IP addresses belonging to these prefixes will more likely to be found in cache. Therefore, the skew in popularity distribution further increases the performance gaps.

When studying the table in more detail, it can be found that when there are 50 FIBs, using a shared data structure with the random network trace doubles the lookup performance, however, the speedup of using a shared data structure with the FUNET trace is about five times higher. The higher speedup of the FUNET trace could be a result of that we used separate parts of the FUNET trace to different virtual routers. Although these parts are separate, the similarity of these parts could be high since they are from the same source. If we used real traces from different sources, the speedup may be lower. However, real traces are very difficult to obtain to perform the study.

6.3 Packet forwarding speed in a Click router

We have also studied the packet forwarding speed in the Click router in the Linux kernel, using the Click configuration similar to the one shown in Fig. 8.

Table 5: Click’s packet forwarding speed (Mbps).

Number of FIBs	2	5	10	20	50
CombinedIPLookup	480	470	470	460	450
RadixIPLookup	460	440	430	420	380

Table 6: Average number of main memory references for forwarding a packet.

Number of FIBs	2	5	10	20	50
CombinedIPLookup	0.0	0.0	0.1	0.2	0.3
RadixIPLookup	0.5	1.0	1.3	1.6	2.4

To avoid the potential bottleneck at the Ethernet interfaces, we have used a *RandomSource* element instead of the *FromDevice* element. The *RandomSource* element is a standard Click element that generates packets with random destination IP addresses. Also, we used a *RoundRobinSwitch* element instead of the *VR-Switch* element. The standard Click element *RoundRobinSwitch* dispatches the packets to different virtual routers in a round robin fashion. Finally, the *ToDevice* element is replaced by the Click element *Discard* that simply drops the packets.

Using this setup, we have studied the packet forwarding speed with different numbers of virtual routers, using the standard *RadixIPLookup* element and our *CombinedIPLookup* element.

Table 5 shows the forwarding speed in megabits per second (Mbps) in different setups. The packet is of size 128 bytes or 1024 bits. As can be seen in the figure, with *CombinedIPLookup*, the forwarding speed varies only slightly, from 480 Mbps to 450 Mbps when the number of virtual routers increases from 2 to 50. Basically, the impact of having a larger number of virtual routers on the forwarding speed is quite limited. However, using the *RadixIPLookup* element, the forwarding speed varies between 460 Mbps and 380 Mbps, and we could notice a more significant performance decrease when running multiple virtual routers.

We also used *oprofile* [29], a system-wide profiler for Linux systems, to study the cache behavior when running the forwarding application in Click. Among the statistic counters for our test computer, there is one counter for number of L2 cache misses, which is also the number of main memory references. As there is no counter for the total number of memory references in our test computer, results in average cache miss rate cannot be provided. In Table 6, the average number of main memory references for forwarding a packet is shown. As can be seen in the table, when using *CombinedIPLookup*, the average number of main memory references is generally smaller than using *RadixIPLookup*. In particular, when using *RadixIPLookup* with 50 virtual routers, there are in average 2.4 main memory references for forwarding each packet, however, when using *CombinedIPLookup*, there are an average of 0.3 references only.

7. CONCLUSION

In this work, we have proposed a shared data structure to efficiently support multiple FIBs in routers. Based on the shared data structure, we have presented an efficient IP-address lookup algorithm. Our shared data structure and algorithm are useful as more and more networking technologies are relying on having multiple FIBs in a physical router. These technologies include for example customer-specific routing, policy-based routing, multi-topology routing and network virtualization. In addition, many recent proposals in networking research rely on having multiple FIBs in a single physical router.

The performance evaluation of the data structure and the algorithm shows that with the proposed data structure, the memory usage is significantly reduced. For example, 4.4 MB is required to store 50 FIBs with a shared data structure, while it requires 75 MB if FIBs are stored separately. In addition, due to smaller data structure size and better cache behavior, the lookup speed is increased significantly in general-purpose processors. When there are two FIBs, the shared data structure nearly doubles the lookup performance compared to an approach with separated FIBs. When there are 50 FIBs, the shared approach leads up to five times speedup.

Our work is the first attempt to solve the emerging problem of efficiently supporting multiple virtual routers. With increasing popularity of virtual routers, the importance of our solution will increase, and we expect to see future research in building scalable routers that support virtualization.

8. REFERENCES

- [1] Juniper, *Intelligent logical router service*, white paper, http://www.juniper.net/solutions/literature/white_papers/200097.pdf, 2004.
- [2] Cisco, *Policy-based routing*, white paper, http://www.cisco.com/warp/public/732/Tech/policy_wp.pdf, 1996.
- [3] P. Psenak, S. Mirtorabi, A. Roy, L. Nguyen and P. Pillay-Esnault, "Multi-Topology (MT) Routing in OSPF", *IETF RFC 4915*, 2007.
- [4] T. Przygienda, N. Shen and N. Sheth, "M-ISIS: Multi Topology (MT) Routing in Intermediate System to Intermediate Systems (IS-IS)", *IETF RFC 5120*, 2008.
- [5] R. Alimi, Y. Wang and Y. R. Yang, "Shadow Configuration as a Network Management Primitive," in Proc. of *ACM SIGCOMM* 2008.
- [6] M. Caesar and J. Rexford, "Building bug-tolerant routers with virtualization", in Proc. of *ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow*, Aug. 2008.
- [7] T. Anderson, L. Peterson, S. Shenker, J. Turner, "Overcoming the Internet impasse through virtualization," *IEEE Computer*, vol.38, no.4, pp. 34-41, April 2005.
- [8] A. Bavier, N. Feamster, M. Huang, L. Peterson, J.

- Rexford. "In VINI Veritas: Realistic and Controlled Network Experimentation," in Proc. of *ACM SIGCOMM*, 2006.
- [9] Route Views Project, University of Oregon. Available: <http://www.routeviews.org/>.
- [10] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083-1092, June 1999.
- [11] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network Magazine*, vol. 15, no. 2, pp. 8-23, Mar. 2001.
- [12] E. Kohler, R. Morris, B. Chen, J. Jannotti and M. F. Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, August 2000.
- [13] A. McAuley and P. Francis, "Fast Routing table Lookups using CAMs," in Proc. of *IEEE Infocom'93*, vol. 3, pp. 1382-1391, San Francisco, 1993.
- [14] F. Zane, N. Giriya and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," in Proc. of *IEEE Infocom'03*, pp. 42-52, San Francisco, May 2003.
- [15] E. Spitznagel, D. Taylor and J. Turner, "Packet Classification Using Extended TCAMs", in Proc. of *ICNP'03*, pp. 120-131, Nov. 2003.
- [16] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Systems*, vol. 17, no. 1, pp. 1-40, Oct. 1999.
- [17] M. Degermark et al., "Small Forwarding Tables for Fast Routing Lookups". in Proc. *ACM SIGCOMM Conference'97*, pages 3-14, Oct. 1997.
- [18] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *IEEE/ACM Transactions on Networking (TON)*, vol. 7, no. 3, pp.324-334, 1999.
- [19] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: scalable IP lookup with fast updates," *Computer Networks: The International Journal of Computer and Telecommunications Networking*, v.44 n.3, p.289-303, Feb. 2004.
- [20] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," in Proc. of *INFOCOM*, vol. 3, 1193-1202, Mar. 2000.
- [21] J. Fu, O. Hagsand and G. Karlsson, "Improving and Analyzing LC-Trie Performance for IP-Address Lookup," *Journal of Networks*, vol. 2, June 2007.
- [22] R. Draves, C. King, S. Venkatachary and B. Zill, "Constructing Optimal IP Routing Tables," in Proc. of *IEEE Infocom*, Mar. 1999.
- [23] E. Fredkin. "Trie memory," *Communications of the ACM*, pp. 490-499. 1960.
- [24] E. Keller and E. Green, "Virtualizing the Data Plane through Source Code Merging", in Proc. of *ACM SIGCOMM Workshop on Programmable Routers for Extensible Services of Tomorrow*, Aug. 2008.
- [25] CSC, Finnish IT Center for Science, FUNET Looking Glass, Available: <http://www.csc.fi/sumoi/funet/noc/looking-glass/lg.cgi>.
- [26] Swedish University Network (SUNET), Available: <http://www.sunet.se> and <http://stats.sunet.se>.
- [27] Emulab, Network Emulation Testbed. Available: <http://www.emulab.net>.
- [28] SPEC, Standard Performance Evaluation Corporation, Available: <http://www.spec.org/>.
- [29] OProfile, Available: <http://oprofile.sourceforge.net/>.