A Trie Merging Approach with Incremental Updates for Virtual Routers

Layong Luo[*†], Gaogang Xie[*], Kavé Salamatian[‡], Steve Uhlig[§], Laurent Mathy[¶], Yingke Xie[*]
[*]Institute of Computing Technology, Chinese Academy of Sciences (CAS), China
[†]University of CAS, China, [‡]University of Savoie, France
[§]Queen Mary, University of London, UK, [¶]University of Liège, Belgium
{luolayong, xie, ykxie}@ict.ac.cn, kave.salamatian@univ-savoie.fr
steve@eecs.qmul.ac.uk, laurent.mathy@ulg.ac.be

Abstract—**Virtual routers are increasingly being studied, as an important building block to enable network virtualization. In a virtual router platform, multiple virtual router instances coexist, each having its own FIB (Forwarding Information Base). In this context, memory scalability and route updates are two major challenges. Existing approaches addressed one of these challenges but not both. In this paper, we present a trie merging approach, which compactly represents multiple FIBs by a merged trie and a table of next-hop-pointer arrays to achieve good memory scalability, while supporting fast incremental updates by avoiding the use of leaf pushing during merging. Experimental results show that storing the merged trie requires limited memory space, e.g., we only need 10MB memory space to store the merged trie for 14 full FIBs from IPv4 core routers, achieving a memory reduction by 87% when compared to the total size of the individual tries. We implement our approach in an SRAM (Static Random Access Memory)-based lookup pipeline. Using our approach, an on-chip SRAM-based lookup pipeline with 5 external stages is sufficient to store the 14 full IPv4 FIBs. Furthermore, our approach can guarantee a minimum update overhead of one write bubble per update, as well as a high lookup throughput of one lookup per clock cycle, which corresponds to a throughput of 251 million lookups per second in the implementation.**

I. Introduction

Network virtualization has recently attracted much interest as it enables the coexistence of multiple virtual networks on a shared physical substrate [1]. The virtual router platform has emerged as a key building block of the physical substrate for virtual networks [2–9]. In a virtual router platform, multiple virtual router instances coexist, each with its own FIB (Forwarding Information Base). With a growing demand for virtual networks, the number of virtual router instances running over a single physical platform and their corresponding FIBs is expected to increase. Generally, it is desirable to store FIBs in high-speed memory to enable high lookup performance. However, the size of high-speed SRAMs (Static Random Access Memory) is limited in router line cards or in general-purpose processors caches. Therefore, supporting as many FIBs as possible in the limited available high-speed memory is becoming a challenge.

With an increasing number of FIBs, more than one FIB is expected to be stored on each high-speed memory chip, and the number of updates to the content of each memory chip becomes the aggregate of the updates of these FIBs. This increases the update frequency, and decreases lookup performance, potentially leading to packet drops, unless fast incremental updates are possible to the multiple FIBs in a single memory chip.

The two above challenges, namely memory scalability and fast incremental updates for virtual routers, have attracted lately some attention in the literature and several approaches have been proposed. However, no previous work has addressed both challenges. For example, [5] proposes a solution that achieves good memory scalability but uses leaf pushing [10] to reduce the node size, which leads to complicated and slow updates [9]. In [8, 9], solutions enabling fast updates fail to achieve good memory scalability. They require almost linear memory increase since they do not actually apply node sharing.

In our previous work [11], we proposed a hybrid IP lookup architecture to address the update challenge, but did not target the memory scalability issue for virtual routers. In this paper, we present a trie merging approach that addresses both of the above challenges simultaneously, *i.e.*, both good memory scalability through trie merging, and fast incremental updates and fast lookups through a lookup pipeline that guarantees a minimum update overhead of one write bubble per update, and a lookup throughput of one lookup per clock cycle. More precisely, the key contributions of this paper are as follows:

1) We propose a new data structure for the nodes of the merged trie, different from the one used in classical trie merging approaches [5]. This new data structure introduces a prefix bitmap that enables the separation of trie nodes and next-hop pointers. This keeps the node size small even when the number of FIBs is large. This data structure avoids leaf pushing during the merging process, and facilitates fast incremental updates.
2) Based on the proposed trie merging approach, we implement an SRAM-based lookup pipeline that guarantees a minimum update overhead of one write bubble per update, as well as a high lookup throughput of one lookup per clock cycle. We implement this lookup pipeline for a virtual router platform with 14 full IPv4 FIBs and evaluate its performance.

FIB 1

prefix	next hop
0*	A1
1*	A2
00*	A3
10*	A4

FIB 2

prefix	next hop
0*	B1
10*	B2
11*	B3
000*	B4

(a)

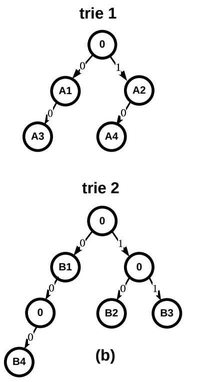

Fig. 1. (a) Two sample FIBs, and (b) their corresponding 1-bit tries

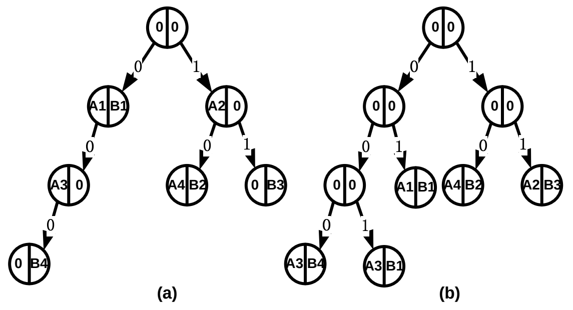

Fig. 2. The merged trie. (a) in trie overlap without leaf pushing and (b) in trie overlap with leaf pushing

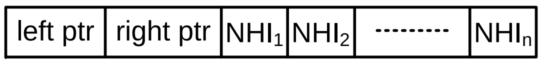

Fig. 3. Node data structure of the merged trie in trie overlap without leaf pushing

The rest of the paper is organized as follows. Section II presents the background and our assumptions. Section III introduces the proposed trie merging approach and its corresponding lookup and update processes. In Section IV, we evaluate the memory requirements. In Section V, we describe the pipelined implementation, and discuss the memory consumption, update overhead and lookup performance. We conclude in Section VI.

II. Background and Assumptions

The literature proposes two types of approaches for storing multiple FIBs in a virtual router platform: separated and merged approaches.

In a separated approach, FIBs are stored in separate memory space with possible different data structures. This approach ensures strong FIBs isolation but at the cost of large memory consumption. By using such an approach, Unnikrishnan *et al.* [12] built virtual routers over the NetFPGA platform [13] with only up to five virtual router instances due to the limited memory in the hardware.

The second type of approach merges FIBs together to improve memory scalability. In [5], an efficient approach is proposed to merge multiple FIBs into a single trie, which we will refer to as *trie overlap*. Unless otherwise stated, the term "*trie*" means a 1-bit trie (*i.e.*, a binary trie) as defined in [14], with the next-hop pointer representing the next hop information for simplicity, and an invalid next-hop pointer being denoted as 0. In trie overlap, first a 1-bit trie is built from each FIB (see Fig. 1). Then, as illustrated in Fig. 2(a), the nodes corresponding to the same prefix in all the tries are merged together in a merged node, and the number of next-hop pointers in each merged node is equal to the number of FIBs (see Fig. 3 for the data structure of a node in the merged trie). As a large number of nodes are shared, the size of the merged trie is smaller than the sum of the size of individual tries.

However, as the number of FIBs increases, the node size increases, leading to large trie size and multiple memory accesses per node visit, degrading lookup and update performance. The authors of [5] proposed to use leaf pushing [10] to reduce the node size in the merged trie, by pushing all the next-hop pointers existing in intermediate nodes to leaf nodes (see for example Fig. 2(b)). After leaf pushing, every intermediate node has two children but no next-hop pointers, and every leaf node has next-hop pointers but no children. Therefore, each node can store only one pointer, pointing to its children for intermediate nodes, or pointing to a next-hop-pointer array containing n next-hop pointers, where n is the number of FIBs, for leaf nodes. This reduces significantly the node size and the trie size. However, leaf pushing makes incremental updates complicated, and a single update may result in reconstructing the entire leaf-pushed trie [15].

In particular, trie overlap works well when the individual FIBs have similar original tries. However, the prefix sets of different FIBs are not always similar, resulting in dissimilar tries. To address this issue, Song *et al.* [6] introduced a trie braiding mechanism that transforms dissimilar tries into similar ones by allowing to swap freely the left and right sub-tries of any node. Nonetheless, Ganegedara *et al.* [7] observed that even by using trie braiding, the original tries built from different Provider Edge (PE) routers are hard to be transformed to similar ones. This is because most prefixes of any PE router have a common portion, which is different among PE routers. They therefore proposed a multiroot approach, which allows tries to be merged at split nodes, rather than only at the root nodes. The sub-tries rooted at the split nodes can be further merged using trie overlap and trie braiding.

Trie overlap, trie braiding and multiroot are all based on node sharing to reduce the memory usage. By contrast, Le *et al.* [8] and Ganegedara *et al.* [9] applied a simple merging approach without node sharing to achieve fast updates. However, the size of the merged tree (or trie) in their approaches increases linearly with the number of prefixes (or tries) [8, 9], resulting in poor memory scalability.

In this paper, we focus on the trie-based merging approach with node sharing. Similarly to trie overlap, we assume that *tries to be merged are similar*. By this, we mean that even if the tries to be merged are dissimilar, these can be transformed into similar ones before merging them, using existing approaches such as trie braiding [6] and multiroot [7]. However, differently from trie overlap that has to use leaf pushing in order to reduce the node size of the merged trie, and significantly increases the update overhead [9], in this paper, we propose a new trie merging approach that does not rely on leaf pushing. In place of leaf pushing, we reduce the node size of the merged trie by introducing a prefix bitmap in each node to separate next-hop pointers from the trie node. As a result, we can build a memory-efficient merged trie that can support fast incremental updates.

III. Proposed Trie Merging Approach

A. Observation

A key observation is that, in trie overlap without leaf pushing, when an IP lookup is visiting a node in the merged trie, the next-hop pointers (*i.e.*, the next-hop-pointer array), contained in it are not necessarily immediately accessed. In fact, an IP lookup only needs to access the next-hop-pointer array of the node corresponding to the longest matched prefix after the trie search terminates. Therefore, it is possible to separate the next-hop-pointer arrays from the merged trie and store them in a different memory.

We illustrate this with an example. Suppose we are looking up for IP address 100 in the merged trie in Fig. 2(a). The nodes <0, 0>, <A2, 0>, and <A4, B2> will be visited in sequence in the lookup. However, the next-hop-pointer array is not necessarily accessed in any one of these nodes during trie search. We only need to access the next-hop-pointer array of node <A4, B2> that corresponds to the longest matched prefix 10* after the trie search terminates. It is therefore not necessary to store these next-hop-pointer arrays in the same memory as the nodes of the merged trie. This observation is the key motivation behind our new data structures.

B. Data Structures

Based on the above observation, we separate the storage of the next-hop-pointer arrays from the merged trie storage, *i.e.*, instead of storing a next-hop-pointer array in each node (see Fig. 3), we only need a simple pointer, pointing to the corresponding next-hop-pointer array in another memory (see Fig. 4). However, we need also to indicate whether the corresponding prefix of the node is a valid prefix for each one of the merged FIBs so that we can find the longest matched prefix. We add a prefix bitmap to each node to indicate the prefix information, *i.e.*, the i^{th} bit ($i \in [1, n]$) of the prefix bitmap is set to 1 when the prefix associated with this node is a valid prefix in the i^{th} FIB, where n denotes the number of FIBs. The resulting data structures of the node in the merged trie and its corresponding next-hop-pointer array are shown in Fig. 4. Each node of the merged trie stores four elements: a left child pointer (*left ptr*), a right child pointer (*right ptr*), an array pointer (*array ptr*) pointing to its corresponding next-hop-pointer array and a prefix bitmap with n-bit size. The next-hop-pointer array is stored in a separate data structure and eventually in a different memory, which contains n next-hop pointers.

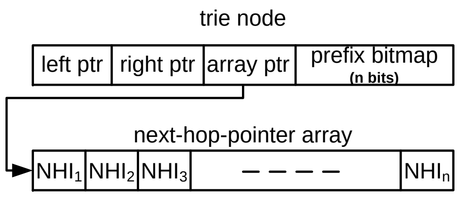

Fig. 4. Data structures of trie node and next-hop-pointer array

Compared to the node data structure in trie overlap without leaf pushing (see Fig. 3), the proposed node data structure scales better in terms of node size. In our approach, with adding one additional trie to merge, only 1 bit of prefix bitmap is added to the nodes of the merged trie, while one next-hop pointer (*e.g.*, 8 bits) has to be added in each node in trie overlap without leaf pushing. Therefore, as the number of tries increases, the node size of our proposed merged trie stays much smaller than that of the merged trie in trie overlap without leaf pushing.

C. Trie Merging Algorithm

In our approach, the trie merging algorithm is similar to the one of trie overlap [5], except that the node data structure of the merged trie is different. Initially, a 1-bit trie is built from each FIB. To distinguish each FIB from each other in the same virtual router platform, a unique virtual router ID (VID) is assigned to each FIB. For merging, the root node of a trie is first merged into the root node of the merged trie. The prefix bitmap in the root node of the merged trie and the corresponding next-hop-pointer array should be modified according to the root node of the trie to be merged and its VID. This algorithm is then called recursively to merge its left sub-trie and right sub-trie, respectively.

The results of applying this merging algorithm to the two tries shown in Fig. 1 are depicted in Fig. 5. The merging process generates a merged trie and a table of next-hop-pointer arrays. To show the difference between trie overlap without leaf pushing and our merging approach, we take the prefix 1* in Fig. 1 as an example. In FIB 1, prefix 1* is a valid prefix and its associated next-hop pointer is A2; in FIB 2, prefix 1* is not a valid prefix. Therefore, in trie overlap without leaf pushing (as shown in Fig. 2(a)), a next-hop-pointer array <A2, 0> should be stored in the corresponding node. In our approach, instead of storing the entire next-hop-pointer array, we only need to store a single pointer P3 pointing to the array <A2, 0> in the table of next-hop-pointer arrays stored in a separate memory. To show whether prefix 1* is a valid prefix in each FIB, a prefix bitmap "1,0" is attached in the corresponding node, where the first bit '1' denotes that prefix 1* is a valid prefix in FIB 1, and the second bit '0' denotes that prefix 1* is not a valid prefix in FIB 2.

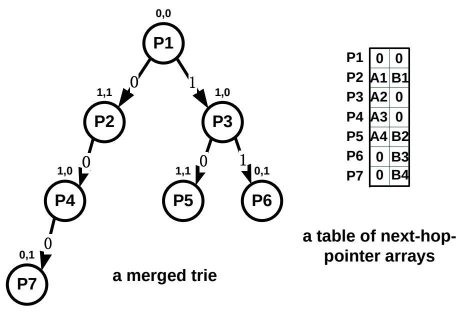

Fig. 5. A merged trie and its corresponding table of next-hop-pointer arrays

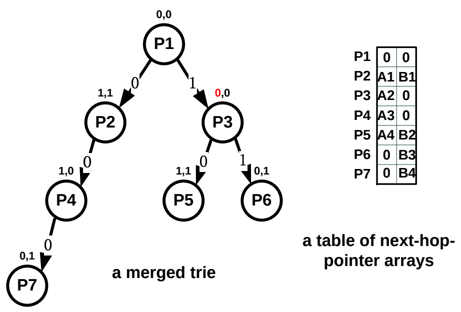

Fig. 6. Deletion of prefix 1* in FIB 1

D. Lookup Process

The lookup process on the merged trie is similar to that on an original 1-bit trie [14]. The major difference is that we check an extra prefix bitmap when accessing each node.

For each incoming packet, the destination IP address and the VID are generated based on its packet header. The destination IP address is used to traverse the merged trie and the VID is used as an index to choose which bit of the prefix bitmap to check. When visiting a node, if the chosen bit of the prefix bitmap is set, the currently visited node is associated with a valid prefix, and its array pointer should be recorded. The recorded array pointer will be updated if the following visited node is also associated with a valid prefix. When the trie lookup terminates, we find the next-hop-pointer array through the array pointer, and extract the VID^{th} next-hop pointer in the array. In this way, a single memory access is needed to find the next-hop pointer after the trie traversal terminates.

For example, suppose the destination IP address of a packet is 100 and its VID is 1 and we are looking up in the merged trie shown in Fig. 5. Nodes P1, P3 and P5 will be accessed in sequence (for simplicity, we use the name of the array pointer in each node to denote the node itself). First, node P1 is accessed and since the first bit position in the prefix bitmap is '0', node P1 is not associated with a valid prefix in FIB 1. Then, we continue to the next node P3, where the corresponding prefix bit in the bitmap is '1' so this node is associated with a valid prefix in FIB 1 and P3 is recorded as the array pointer. We continue to node P5 that is also associated with a valid prefix in FIB 1 as its corresponding prefix bit is '1'. Therefore, P5 is recorded as a new array pointer. Since node P5 is a leaf node, the trie traversal terminates and we use the latest array pointer P5 and the index 1 from VID to get the next-hop pointer A4 from the table of next-hop-pointer arrays.

E. Update Process

In terms of route updates, the original 1-bit trie is an excellent data structure [14, 15], as at most a single prefix should be modified for each route update. By avoiding leaf pushing during merging, we keep the merged trie very similar to the original 1-bit trie. In our approach, the updates are processed similarly to the original 1-bit trie, and at most one prefix should be modified for each route update.

For example, if we delete prefix 1* from FIB 1 shown in Fig. 1, the corresponding modification in the merged trie is shown in Fig. 6. First, we use prefix 1* to traverse the trie, and find the node corresponding to prefix 1* (node P3). Then, VID 1 is used to choose the first bit of the prefix bitmap in node P3 and this bit is reset to '0', which makes prefix 1* invalid for FIB 1. Note that the corresponding next-hop-pointer array does not need to be modified in this case.

The insertion of a new prefix and the modification of an existing prefix can be implemented similarly with a single prefix change. The only difference is that, for insertions and modifications, the table of next-hop-pointer arrays should also be modified. In these cases, one additional memory access to the table of next-hop-pointer arrays is needed after the update on the merged trie terminates.

IV. Evaluation of Memory Usage

In this section, we compare the memory requirements of four approaches: separated approach, which stores each trie separately without node sharing, trie overlap without leaf pushing, trie overlap with leaf pushing, and our proposed approach. The lookup and update performance will depend on implementation details and will be evaluated in the implementation section.

A. Routing Tables

As router virtualization is still not widely used, limited data is available. Instead, we use publicly available BGP routing data and assume that these routing tables are used for virtual routers. We rely on 14 full IPv4 BGP routing tables collected from the RIPE RIS Project [16]. We extracted for each table the unique prefixes and formed a FIB for it. A 1-bit trie was built for each of them. We list in Table I the statistics of the routing tables and the resulting tries.

B. Number of Trie Nodes

We show for the four techniques compared, in Fig. 7(a) the total number of trie nodes in the merged trie as a function of the number of virtual routers. As expected for the separated approach, the total number of trie nodes increases linearly as the number of virtual routers increases. Trie overlap without leaf pushing and our proposed approach have the same number of nodes in the merged trie as the prefix sets similarities are

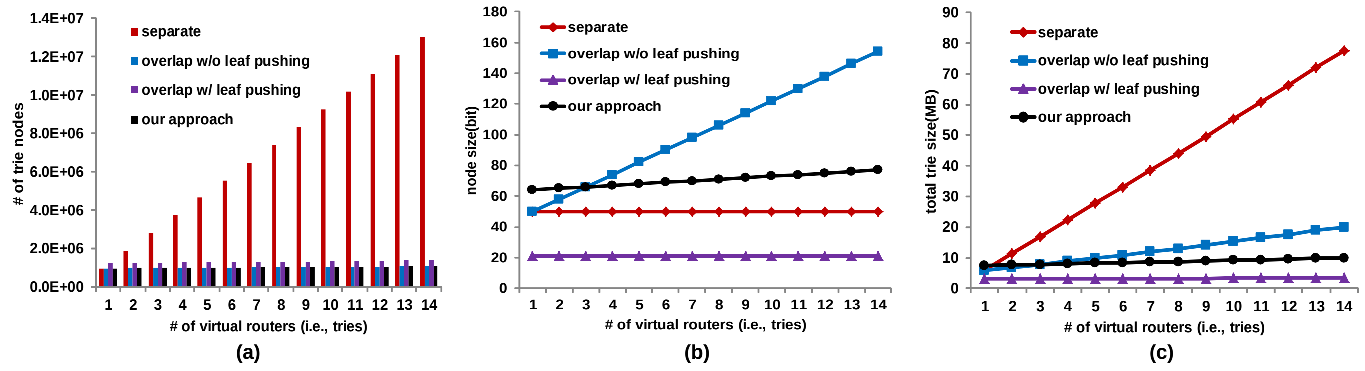

Fig. 7. Memory comparison. (a) number of nodes, (b) node size, and (c) trie size

TABLE I
ROUTING TABLES (2011.09.29, 08:00)

Router	Location	# of prefixes	# of nodes	# of updates	
				Announce	Withdraw
rrc00	Amsterdam	399,439	974,050	3,480,740	325,553
rrc01	London	375,751	918,417	2,040,708	227,723
rrc03	Amsterdam	373,306	913,538	1,418,791	1,416,709
rrc04	Geneva	382,122	936,888	768,336	59,345
rrc05	Vienna	375,196	915,995	1,476,029	218,630
rrc06	Otemachi	367,984	898,119	60,522	7,275
rrc07	Stockholm	379,788	927,129	572,376	36,230
rrc10	Milan	373,024	910,537	291,679	191,079
rrc11	New York	379,166	926,368	1,251,304	219,523
rrc12	Frankfurt	386,924	947,351	2,436,235	318,413
rrc13	Moscow	381,561	935,345	2,499,320	192,601
rrc14	Palo Alto	380,048	926,692	1,175,918	82,797
rrc15	Sao Paulo	392,537	957,129	5,275,719	397,876
rrc16	Miami	382,552	935,854	17,886	1,150

exploited in the same way. For these two approaches the total number of nodes increases slowly. This can be explained by observing that as the number of tries increases, the merged trie become denser and more nodes are shared. After merging 14 tries, the merged trie contains 1,083,217 nodes, which are just 11% more nodes than that in the largest individual trie (trie relative to the rrc0 router). The behaviour for trie overlap with leaf pushing is similar with a slow increase of the number of nodes. However, leaf pushing adds some extra nodes and the number of nodes becomes larger than that in trie overlap without leaf pushing and our proposed approach.

C. Node Size

We can also compare the storage size of each node. We have used 21 bits to store the left or right child pointer (a 21-bit pointer can represent 2,097,152 nodes, which is larger than the total number of nodes in the merged trie after 14 tries are merged), 8 bits to store the next-hop pointer and 21 bits for the array pointer. The node sizes are shown in Fig. 7(b). The node size of each individual node remains the same at 50 bits in the separated approach and does not depend on the number of virtual routers. The node size in trie overlap with leaf pushing also stays the same at 21 bits, as each node only contains a single pointer (as mentioned in Section II). In trie overlap without leaf pushing, every added trie leads to an 8-bit (the size of a next-hop pointer) increase in the node size, since one next-hop pointer has to be added into the node for every added trie. In our approach, every added trie leads to only a 1-bit increase in the node size, since only 1 bit needs to be added in the prefix bitmap.

D. Total Trie Size

The total trie size is determined by the total number of nodes and the node size, which we have already evaluated in the above two sub-sections. We now evaluate the total trie size. The results are shown in Fig. 7(c).

Trie overlap with leaf pushing scales best in terms of trie size, as one node only needs to contain a single pointer. Our approach can achieve a significant memory saving when compared to the separated approach and the trie overlap without leaf pushing. We need only 10MB memory space for merging 14 tries and achieve a memory reduction by 87% and 50% respectively, when compared to the separated approach and trie overlap without leaf pushing. Moreover, the growth trend of the total trie size shows that, the memory requirement in our approach increases more slowly than that of those two approaches. This makes our approach a scalable solution for virtual routers in terms of high-speed memory requirements.

E. Size of the Table of Next-hop-pointer Arrays

In both our approach and trie overlap with leaf pushing, the next-hop pointers are separated from the merged trie. Therefore, an extra table of next-hop-pointer arrays is needed. However, as only a single memory access is needed to this table after the end of trie traversal, this table can be stored in a relatively slow and large external memory, making the size of this table less important. In our experiments, after merging 14 tries, the number of nodes becomes 1,083,217, and thus the size of the table of next-hop-pointer arrays is about 14MB.

F. Size of the Complete Next Hops

We also need memory space to store the complete next hop information that consists of the IP address of the next hop

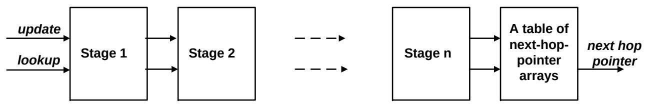

Fig. 8. A native pipeline for our merged trie

and the corresponding output interface. Most routers have less than 40 output interfaces, so the total size of the next hop information for a FIB is usually below 200 bytes [5], and is the same for all approaches. Therefore, we will ignore the complete next hops in the rest of this paper.

V. Pipelined Implementation

The SRAM-based lookup pipeline [17] is a very good implementation for trie-based data structures, as it scales well in terms of lookup throughput [18]. In this section, we implement an SRAM-based lookup pipeline for scalable virtual routers, and compare the four approaches.

A. Native Pipeline

A native way for pipelining is to map each trie level into a different pipeline stage, each with its own SRAM [19, 20]. For example, in Fig. 5, there are four trie levels: node P1 in level 1, node P2, P3 in level 2, node P4, P5, P6 in level 3 and node P7 in level 4. In a native pipeline, each stage contains the nodes in one of the four levels. Fig. 8 shows the native pipeline in our approach. It consists of n stages, where n is the number of levels in the merged trie. In the i^{th} stage, all the nodes in the i^{th} level of the trie are stored. As we separate the next-hop-pointer arrays from the trie nodes, an additional pipeline stage should be added at the end of the pipeline to store the table of the next-hop-pointer arrays. For an IP lookup, it enters into the pipeline at stage 1, goes through the pipeline by accessing at most one node in each stage, and exits after the next-hop pointer is obtained at the final stage. As each pipeline stage has its own separate SRAM, all stages can be visited simultaneously. As a result, a new IP lookup can be issued into the pipeline after the preceding IP lookup moves to stage 2. If each node can be accessed in just one clock cycle, the SRAM-based lookup pipeline can achieve a lookup throughput of one lookup per clock cycle. Route updates in the pipeline are performed in the form of write bubbles [19]. A write bubble also goes through the pipeline by accessing at most one node in each stage, at the same speed as the lookup. The difference between a write bubble and an IP lookup is that, a write bubble performs at most one write operation in each stage, while an IP lookup always performs read operations.

B. Memory Issues in Practical Pipelines

In practice, the native pipeline suffers from two memory issues.

The first issue is the memory size. In the pipeline, a large number of separate SRAMs are required, as each pipeline stage has its own SRAM. For example, in the case of IPv4, we need 33 separate SRAMs in the native pipeline. It is impractical to have so many separate SRAM chips in router

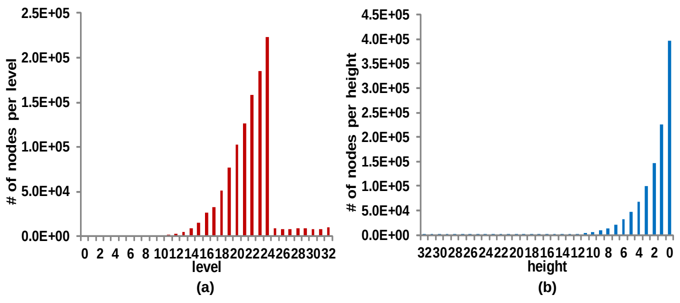

Fig. 9. (a) Node distribution based on trie level, and (b) node distribution based on trie height

line cards. FPGA is a natural candidate to implement an SRAM-based pipeline, as it contains a large number of on-chip SRAMs. For example, a large FPGA from Xilinx [21] contains over 100 on-chip SRAMs. However, the total size of on-chip SRAMs within the FPGA is too small to store a full IPv4 FIB. For example, the total size of on-chip SRAMs in large Virtex-6 FPGAs [21] is from 5Mb to 37Mb, while a single 1-bit trie built from a full IPv4 FIB is 5.8MB (*i.e.*, 46.4Mb), see Fig. 7(c). The situation gets worse in the context of virtual routers, where multiple FIBs exists. In order to solve the memory size issue of the SRAM-based lookup pipeline within the FPGA, large external SRAMs are required. These large external SRAMs can be added into the pipeline in two ways. First, the few largest levels of the trie can be moved to external SRAMs [22]. Fig. 9(a) shows the node distribution of the merged trie for 14 full IPv4 FIBs in our approach based on trie levels. The few levels around level 24 contain many more nodes than other levels, and thus these stages can be moved to external SRAMs to significantly reduce the memory requirement of the on-chip SRAM-based pipeline. We will call this approach "*level-based partitioning*". We propose another way for adding external stages, by moving trie nodes based on trie height to external SRAMs. For example, all the leaf nodes (*i.e.*, nodes with height 0) in Fig. 5 can be moved to an external stage. More stages can be moved to external SRAMs by removing leaf nodes repeatedly. Fig. 9(b) shows the node distribution based on trie height. The few stages around height 0 contain many more nodes than other stages. Therefore, moving these few stages to external SRAMs can also significantly reduce the memory requirement of the on-chip SRAM-based pipeline. We will call this approach "*height-based partitioning*". Fig. 10 shows the ratio of the remaining nodes in the on-chip pipeline after a few stages are moved to external stages. We observe that after moving a given number of stages to external SRAMs, the number of nodes left in on-chip SRAMs when using height-based partitioning is smaller than that when using level-based partitioning. Therefore, we will adopt height-based partitioning to solve the memory size issue of the on-chip SRAM-based lookup pipeline within the FPGA.

The second memory issue of the native pipeline is the memory distribution. For level-based mapping, the node distribution

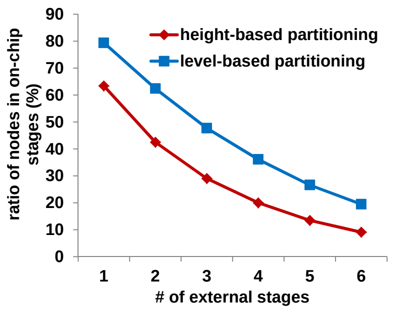

Fig. 10. Ratio of nodes remaining in the on-chip pipeline

of the merged trie for the 14 FIBs is shown in Fig. 9(a). The number of nodes in each stage varies substantially, leading to inefficient memory utilization [19]. One solution is to assign the equal-size SRAMs to each stage, and then balance nodes across stages. Jiang *et al.* [20] proposed an approach called OLP, which achieved balanced memory distribution across stages. In our implementation, we adopt OLP to balance memory across on-chip pipeline stages within the FPGA.

The goal of our implementation is to be able to map the 14 full IPv4 FIBs shown in Table I into an SRAM-based lookup pipeline. The Xilinx FPGA XC6VLX240T containing 416 36Kb on-chip SRAMs, is used to build the on-chip SRAM-based lookup pipeline. In our trie merging approach, about 80Mb memory is required to store the merged trie for the 14 full FIBs (see Fig. 7(c)). Therefore, less than 18% of the total nodes should be left within the FPGA. Using the height-based partitioning, we find that, after only 4 external stages are added, the remaining nodes can fit in the on-chip SRAMs within the given FPGA. We use OLP [20] to balance the remaining nodes and build a 24-stage on-chip SRAM-based pipeline. The number of nodes in each stage is almost the same (about 8,625 nodes), and 407 on-chip 36Kb SRAMs are needed in total, which is less than the total number of on-chip SRAMs within the FPGA. Therefore, with the given FPGA and 4 external stages (*i.e.*, 4 external SRAMs), the merged trie for 14 full IPv4 FIBs in our approach can be mapped into an SRAM-based lookup pipeline. Additionally, one more external stage for the table of next-hop-pointer arrays is required.

In the similar way, we map the tries in the other three approaches into the SRAM-based pipeline within the same FPGA. The number of external stages required in the pipeline using the four approaches are summarized in Table II. These results are consistent with those shown in Fig. 7(c). A smaller trie can be mapped into the SRAM-based pipeline within the FPGA with fewer external stages. Note that, the results in our approach and in trie overlap with leaf pushing include an external stage for the table of next-hop-pointer arrays.

C. Update Overhead

Although trie overlap with leaf pushing scales best in terms of memory requirement, its update overhead is much higher than that in the other three approaches, as leaf pushing makes route updates complicated. In this section, we will evaluate the update overhead in the pipeline based on all the four approaches. The number of write bubbles per update can be used as the metric to evaluate the update overhead in an SRAM-based IP lookup pipeline [19].

TABLE II
NUMBER OF EXTERNAL STAGES

Approach	# of external stages
separate	9
overlap w/o leaf pushing	6
overlap w/ leaf pushing	3
our approach	5

TABLE III
NUMBER OF WRITE BUBBLES PER UPDATE (THEORETICAL BOUNDS)

Approach	Maximum	Minimum
overlap w/ leaf pushing	2^{W-1}	0
the other three approaches	1	1

In trie overlap with leaf pushing, the number of write bubbles caused by one route update largely depends on where the update occurs. For example, if prefix <1*, B5> is added in FIB 2 shown in Fig. 1, it changes nothing in the leaf-pushed merged trie shown in Fig. 2(b) and no write bubbles are needed for this update. If a route update changes the prefix <000*, B4> in FIB 2 to <000*, B5>, only one node in the leaf-pushed merged trie should be modified, which requires only one write bubble. However, if a route update changes prefix <00*, A3> in FIB 1 to <00*, A5>, two nodes in the leaf-pushed merged trie should be modified. Since these two nodes are in the same stage of the pipeline, two write bubbles are needed for this update. Note that a write bubble can modify at most one node in each stage [19]. In the theoretical worst case, one route update might change at most 2^{W-1} nodes in the same stage, leading to 2^{W-1} write bubbles for this update, where W is the maximum length of the IP prefix. In the other three approaches, leaf pushing and any other prefix expansion technologies are not used. As a result, the tries in the other three approaches are similar to the original 1-bit trie, and at most one node in each level should be modified for one route update, which guarantees that one write bubble is required for one route update in any case. The theoretical comparison of update overhead is summarized in Table III.

To evaluate the update overhead in practice, we obtained 12-hour BGP update traces on all 14 full FIBs shown in Table I. The number of prefix announcements and withdrawals are shown in the last two columns of Table I, respectively. These traces contain more than 26 million updates in total. Fig. 11 shows the complementary cumulative distribution of the update overhead in trie overlap with leaf pushing based on the 12-hour BGP update traces. The number of write bubbles per update varies significantly, as the overhead of each update is largely dependent on where the update occurs on the merged trie. During the 12 hours of the traces, there are 892 updates, *i.e.*, around 0.003% of the total updates, require more than 100

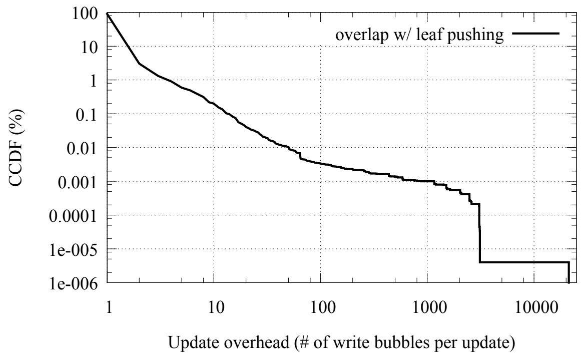

Fig. 11. Complementary cumulative distribution function (CCDF) of actual update overhead in trie overlap with leaf pushing

TABLE IV
NUMBER OF WRITE BUBBLES PER UPDATE IN PRACTICE

Approach	Maximum	Minimum
overlap w/ leaf pushing	21,200	0
the other three approaches	1	1

write bubbles per update in trie overlap with leaf pushing. Although these do not occur often, once they happen, such updates block the lookup pipeline for many clock cycles, leading to many packet drops unless very large packet buffers can store the incoming packets during these cycles. Table IV summarizes the update overhead in the four approaches based on the 12-hour update traces. In trie overlap with leaf pushing, the worst-case update overhead is 21,200 write bubbles per update. In this case, consider a link rate of 100Gbps (*i.e.*, about 5.12ns per packet time for 64-byte packets) and a clock frequency of 250MHz for the pipeline, as many as 16,563 packets may have to be buffered for this update to avoid packet drops. Therefore, the high worst-case update overhead in trie overlap with leaf pushing may impose a significant burden on router buffers. In the other three approaches (*i.e.*, the separated approach, trie overlap without leaf pushing and our approach, the update overhead is one write bubble per update in any case, which is consistent with the result shown in Table III.

D. Lookup Performance Considerations

Generally speaking, for pipelined implementations, all the four approaches can achieve the same lookup throughput of one lookup per clock cycle. However, some subtle differences exist due to the different node size.

For a given SRAM entry size, the lookup throughput of the pipeline is affected by the node size of the trie. If the node size is smaller than the SRAM entry size, a trie node can fit in just one SRAM entry. As a result, one node access can be completed in just one clock cycle, and a throughput of one lookup per cycle can be achieved in the pipeline. However, if the node size is larger than the SRAM entry size, multiple SRAM entries are required to store one node, and thus multiple memory accesses are needed for visiting one node. In this case, one lookup per clock cycle cannot be guaranteed. Therefore, it is easier to achieve a lookup throughput of one lookup per clock cycle if the node size is smaller.

Among the four approaches, trie overlap without leaf pushing scales worst in terms of node size, as a next-hop pointer (8 bits) should be added into the node of the merged trie for every added trie, as shown in Fig. 7(b). That means, as the number of FIBs increases, the node size in trie overlap without leaf pushing is easier to exceed the given SRAM entry size, which may lead to degradation of lookup performance. Note that the node size also affects the performance of write bubbles in the same way.

Fortunately, in our implementations, the trie node in all the four approaches for 14 full IPv4 FIBs, can fit in one SRAM entry within the FPGA, as multiple on-chip SRAMs can be combined together to form a large SRAM entry. As a result, a throughput of one lookup per clock cycle can be achieved. We have implemented the pipeline for our approach within the FPGA, and the results show a maximum clock frequency of 251MHz, which means that the pipeline can achieve a throughput of 251 million lookups per second. All the other three approaches can also achieve a comparable lookup throughput in pipelined implementations.

E. Summary

Through the pipelined implementation, the following conclusions can be drawn.

(1) **Separated Approach**. For SRAM-based ip lookup pipelines, the separated approach is good in terms of lookup and update performance. However, the total size increases drastically when the number of virtual routers increases, which poses a great memory scalability challenge on the SRAM-based lookup pipeline. We have shown that, for 14 full IPv4 FIBs, about 9 external stages are needed to reduce the memory requirement for SRAM-based pipeline within the FPGA.

(2) **Our Approach vs. Trie Overlap without Leaf Pushing**. Our approach is an improvement for trie overlap without leaf pushing, and the main difference is that, the node size in our approach scales much better than that in trie overlap without leaf pushing. Small nodes in our approach benefit the pipeline in two aspects. First, it is easier for a trie with smaller nodes to achieve higher lookup and update performance, as mentioned in Section V.D. Second, the smaller trie size poses a smaller memory size challenge on the SRAM-based lookup pipeline within the FPGA, and thereby fewer external stages are needed in our approach.

(3) **Our Approach vs. Trie Overlap with Leaf Pushing**. Trie overlap with leaf pushing outperforms our approach in terms of memory scalability. However, the main drawback in trie overlap with leaf pushing is the large worst-case update overhead, which may pose a great challenge on router buffer design. By contrast, our approach makes a good balance between the memory scalability and the update overhead. We can achieve good memory scalability and guarantee fast incremental updates simultaneously. Therefore, our approach is a viable alternative to trie overlap with leaf pushing for virtual routers.

VI. Conclusion

In this paper, we proposed a trie merging approach with fast updates to address both memory scalability and route updates challenges for virtual routers. We introduced a prefix bitmap in each node of the merged trie to separate next-hop pointers from trie nodes, which brings scalability to node size and therefore the whole merged trie size. Moreover, through the prefix bitmap, we completely avoided leaf pushing to reduce the node size, and by this enabled fast incremental updates. We implemented an SRAM-based lookup pipeline for the merged trie. We proposed an efficient approach called height-based partitioning, to address the memory size issue of the on-chip SRAM-based pipeline. We evaluated how leaf pushing behaves with real update traces and showed that leaf pushing leads to high worst-case update overhead in practice. Based on our approach, an on-chip SRAM-based lookup pipeline with 5 external stages can store 14 full IPv4 FIBs, and guarantee a low update overhead of one write bubble per route update, as well as a high lookup throughput of one lookup per clock cycle.

The relatively small size of the merged trie in our approach, *e.g.*, 10MB (see Fig. 7(c)), suggests that the trie processing can be done mainly in the cache memory of modern processors. This opens opportunities to exploit the massive parallelism available in modern multi-core or many-core processors, to achieve good memory scalability, fast lookups and fast updates for virtual routers.

Acknowledgment

This work was supported in part by National Basic Research Program of China with Grant 2012CB315801, National Natural Science Foundation of China (NSFC) with Grants 61133015, 61202411 and 61061130562, and Strategic Priority Research Program of the Chinese Academy of Sciences with Grant XDA06010303.

References

[1] N. M. M. K. Chowdhury and R. Boutaba, “A survey of network virtualization,” *Computer Networks*, 2010.

[2] G. Xie, P. He, H. Guan, Z. Li, Y. Xie, L. Luo, J. Zhang, Y. Wang, and K. Salamatian, “PEARL: a programmable virtual router platform,” *IEEE Communications Magazine*, 2011.

[3] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, “Towards high performance virtual routers on commodity hardware,” in *Proc. ACM CoNEXT*, 2008.

[4] M. B. Anwer, M. Motiwala, M. bin Tariq, and N. Feamster, “Switchblade: a platform for rapid deployment of network protocols on programmable hardware,” in *Proc. ACM SIGCOMM*, 2010.

[5] J. Fu and J. Rexford, “Efficient IP-address lookup with a shared forwarding table for multiple virtual routers,” in *Proc. ACM CoNEXT*, 2008.

[6] H. Song, M. Kodialam, F. Hao, and T. Lakshman, “Building scalable virtual routers with trie braiding,” in *Proc. IEEE INFOCOM*, 2010.

[7] T. Ganegedara, W. Jiang, and V. Prasanna, “Multiroot: Towards memory-efficient router virtualization,” in *Proc. IEEE ICC*, 2011.

[8] H. Le, T. Ganegedara, and V. K. Prasanna, “Memory-efficient and scalable virtual routers using FPGA,” in *Proc. FPGA*, 2011.

[9] T. Ganegedara, H. Le, and V. K. Prasanna, “Towards On-the-Fly Incremental Updates for Virtualized Routers on FPGA,” in *Proc. FPL*, 2011.

[10] V. Srinivasan and G. Varghese, “Fast address lookups using controlled prefix expansion,” *ACM Transactions on Computer Systems*, 1999.

[11] L. Luo, G. Xie, Y. Xie, L. Mathy, and K. Salamatian, “A hybrid IP lookup architecture with fast updates,” in *Proc. IEEE INFOCOM*, 2012.

[12] D. Unnikrishnan, R. Vadlamani, Y. Liao, A. Dwaraki, J. Crenne, L. Gao, and R. Tessier, “Scalable network virtualization using FPGAs,” in *Proc. FPGA*, 2010.

[13] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, “NetFPGA - an open platform for teaching how to build gigabit-rate network switches and routers,” *IEEE Transactions on Education*, 2008.

[14] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, “Survey and taxonomy of IP address lookup algorithms,” *IEEE Network*, 2001.

[15] W. Eatherton, G. Varghese, and Z. Dittia, “Tree bitmap: Hardware/software IP lookups with incremental updates,” *Computer Communication Review*, 2004.

[16] RIPE RIS Raw Data. [Online]. Available: http://www.ripe.net/data-tools/stats/ris/ris-raw-data

[17] S. Sikka and G. Varghese, “Memory-efficient state lookups with fast updates,” in *Proc. ACM SIGCOMM*, 2000.

[18] J. Hasan and T. N. Vijaykumar, “Dynamic pipelining: Making IP-lookup truly scalable,” in *Proc. ACM SIGCOMM*, 2005.

[19] A. Basu and G. Narlikar, “Fast incremental updates for pipelined forwarding engines,” in *Proc. IEEE INFOCOM*, 2003.

[20] W. Jiang and V. K. Prasanna, “A memory-balanced linear pipeline architecture for trie-based IP lookup,” in *Proc. IEEE HOTI*, 2007.

[21] Xilinx Inc. [Online]. Available: http://www.xilinx.com/

[22] W. Jiang and V. Prasanna, “Towards practical architectures for SRAM-based pipelined lookup engines,” in *Proc. IEEE INFOCOM Workshops*, 2010.