This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

A Hybrid Hardware Architecture for High-Speed IP Lookups and Fast Route Updates

Layong Luo, Gaogang Xie, *Member, IEEE*, Yingke Xie, Laurent Mathy, *Member, IEEE*, and Kavé Salamatian, *Member, IEEE*

***Abstract*—As network link rates are being pushed beyond 40 Gb/s, IP lookup in high-speed routers is moving to hardware. The ternary content addressable memory (TCAM)-based IP lookup engine and the static random access memory (SRAM)-based IP lookup pipeline are the two most common ways to achieve high throughput. However, route updates in both engines degrade lookup performance and may lead to packet drops. Moreover, there is a growing interest in virtual IP routers where more frequent updates happen. Finding solutions that achieve both fast lookup and low update overhead becomes critical. In this paper, we propose a hybrid IP lookup architecture to address this challenge. The architecture is based on an efficient trie partitioning scheme that divides the forwarding information base (FIB) into two prefix sets: a large disjoint leaf prefix set mapped into an external TCAM-based lookup engine and a small overlapping prefix set mapped into an on-chip SRAM-based lookup pipeline. Critical optimizations are developed on both IP lookup engines to reduce the update overhead. We show how to extend the proposed hybrid architecture to support virtual routers. Our implementation shows a throughput of 250 million lookups per second (equivalent to 128 Gb/s with 64-B packets). The update overhead is significantly lower than that of previous work, the memory consumption is reasonable, and the utilization ratio of most external TCAMs is up to 100%.**

***Index Terms*—IP lookup, route updates, ternary content addressable memory (TCAM), static random access memory (SRAM)-based pipeline.**

I. INTRODUCTION

IN INTERNET routers, IP lookup is a critical function which determines how to forward an incoming packet by finding the next hop for the destination IP address of the packet. Since the introduction of classless inter-domain routing (CIDR) in 1993, finding the next hop for a destination IP address has become a longest-prefix matching (LPM) problem, i.e., given a destination IP address, multiple matching IP address prefixes of different lengths may exist in the forwarding information base (FIB) of the router, and the longest such prefix must be used to determine the next hop.

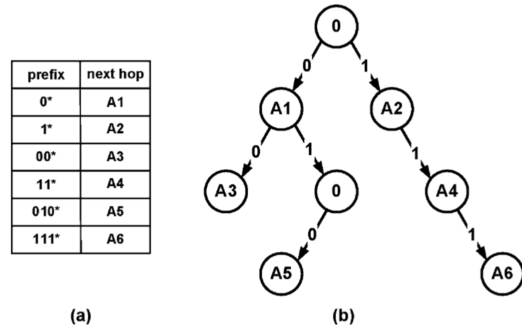

Fig. 1. (a) Sample FIB and (b) corresponding 1-bit trie.

The longest-prefix matching problem naturally lends itself to a hierarchical data structure for which a trie is an efficient representation. An IP lookup trie contains two types of nodes: 1) *prefix nodes* that represent predefined prefixes for which valid next-hop information exists; and 2) *non-prefix nodes* that do not contain next-hop information. Fig. 1 shows a 1-bit trie built from a sample FIB. For simplicity, we use a next-hop pointer to represent the next-hop information, and a "0" denotes an invalid next-hop pointer (i.e., the node containing a "0" is a non-prefix node).

With this trie data structure, the address space represented by the prefix stored at a node is always contained within the address space represented by the prefix stored at its ancestor nodes. The longest-prefix matching of a destination address is then determined by following a single path from the trie root, with the longest-prefix match corresponding to the last prefix node encountered before the end of the path. As there is only one leaf node per trie-path, prefixes stored at different leaf nodes are disjoint, i.e., the corresponding address spaces of two leaves have no address in common.

As network link rates are being pushed beyond 40 Gb/s, IP lookup with LPM becomes a major bottleneck in high-speed routers. The high performance required by such high link rates is hard to be achieved in software [1], and two major hardware implementation techniques have been widely used to achieve such high performance: ternary content addressable memory (TCAM)-based IP lookup engines and static random access

Manuscript received March 02, 2012; revised March 07, 2013; accepted May 21, 2013; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. Crowley. This work was supported in part by the National Basic Research Program of China under Grant 2012CB315801, the NSFC under Grant 61133015, the NSFC-ANR pFlower project under Grant 61061130562, the National High-tech R&D Program of China under Grant 2013AA013501, and the Strategic Priority Research Program of CAS under Grant XDA06010303. The work of L. Mathy was supported in part by Lancaster University. *(Corresponding author: G. Xie)*

L. Luo is with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: luolayong@ict.ac.cn).

G. Xie and Y. Xie are with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: xie@ict.ac.cn; ykxie@ict.ac.cn).

L. Mathy is with the University of Liège, 4000 Liège, Belgium (e-mail: laurent.mathy@ulg.ac.be).

K. Salamatian is with University of Savoie, 74016 Annecy-Le-Vieux, France (e-mail: kave.salamatian@univ-savoie.fr).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2013.2266665

1063-6692/$31.00 © 2013 IEEE

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

memory (SRAM)-based IP lookup pipelines. Both of these solutions can achieve a high throughput of one lookup per clock cycle.

However, both of these solutions suffer from practical problems. On one hand, TCAM-based IP lookup engines have very high update cost because of overlapping prefixes. On the other hand, SRAM-based IP lookup pipelines cannot accommodate large FIBs common in typical situations due to the limited size of on-chip SRAMs in the field-programmable gate array (FPGA), which is a natural hardware choice for implementing SRAM-based lookup pipelines. Additionally, route updates in previous SRAM-based IP lookup pipelines may lead to disruption to the IP lookup process and degrade IP lookup performance.

In this paper, we mainly target fast incremental route updates in high-speed routers. We propose a different view to the problem of hardware IP lookup engine design. Rather than using only one type of hardware solution, TCAM-based IP lookup engine or SRAM-based IP lookup pipeline, we mix these two in order to benefit from the positive points of each approach without being hindered by their weaknesses. Our aim is to design a very fast lookup architecture that enables fast updates concomitantly. For this purpose, we propose a hybrid lookup architecture, composed of a TCAM-based lookup engine and an SRAM-based lookup pipeline operating in parallel. An efficient trie partitioning scheme is applied to partition the FIB of the router into a large disjoint prefix set and a small set of prefixes that overlap those in the disjoint prefix set (we call this latter "*overlapping prefix set*" for short). The TCAM-based lookup engine contains the disjoint prefix set, and the SRAM-based lookup pipeline contains the overlapping prefix set. We show that this hybrid architecture results in fast lookup combined with easy and fast updates. We also show how our approach can be applied in the context of virtual routers.

We implement the proposed hybrid architecture on our PEARL hardware platform [2] and achieve a maximum throughput of 250 million lookups per second (MLPS). Comparative results show that the update overhead is significantly lower than that of previous work. Moreover, the memory consumption in our architecture is reasonable, and our TCAM memory can easily be dimensioned to achieve memory space utilization close to 100%.

The rest of the paper is organized as follows. In Section II, we present the state of the art and explain our design motivation and rationale. In Section III, we introduce our hybrid architecture and describe the optimizations for fast updates. In Section IV, we extend our approaches to support virtual routers. In Section V, we describe the architecture implementation on our PEARL platform and compare its performance to other techniques. We discuss some extensions in Section VI and conclude the paper in Section VII.

II. Background and Motivation

In this section, we discuss two major hardware techniques for implementing fast IP lookup, their practical problems, and related work on these problems. We then explain our design motivation and rationale.

A. TCAM-Based IP Lookup Engines

A TCAM implements a high-speed associative memory, where in a single clock cycle a search key is compared simultaneously with all the entries (i.e., keys) stored to determine a match and output the corresponding address. As TCAM entries can be specified using three states (0, 1, and "X" meaning don't care), this type of memory is particularly well suited for storing IP prefixes where masked bits are given "X" states. Indeed, because of the "X" bits, several TCAM entries could match a given IP address, so TCAMs are designed to always return the first matching entry encountered (TCAM entries have an intrinsic order represented by an address). Therefore, in order to provide correct LPM operations, prefixes are stored in the TCAM with reverse order in overlap, i.e., longest prefix should be stored first. These order constraints result in a large number of TCAM entry movements on some route updates, with large impact on the lookup performance and possible packet drops [3].

Because of the interest of the TCAM for IP lookup, several research efforts have targeted the issue of TCAM updates. In [4], two approaches named PLO_OPT and CAO_OPT have been proposed. PLO_OPT maintains the prefix-length order by putting all the prefixes in order of decreasing prefix lengths and keeping the unused (i.e., empty) space in the center of a TCAM. CAO_OPT relaxes the constraint to only overlapping prefixes in the same chain (i.e., a single path from the trie root). Both approaches can decrease the number of entry movements per update. However, multiple entry movements are still needed for a single route update in the worst case [4]. In another approach, order constraints can be totally avoided in a TCAM by converting the whole prefix set into an equivalent minimum independent prefix set (MIPS) [5] using the leaf pushing technique [6]. However, leaf pushing may lead to prefix expansion. When a prefix is updated, all of its expanded prefixes have to be modified. Therefore, multiple write accesses may still be needed for a single route update. Additionally, TCAM updates can be performed without packet drops by duplicating the TCAM, with updates done to the shadow TCAM and the active one swapped out. However, the TCAM memory requirements are double.

B. SRAM-Based IP Lookup Pipelines

The other major hardware implementation technique, which can also achieve a high throughput of one lookup per clock cycle, is the SRAM-based lookup pipeline [7], which corresponds to a straightforward mapping of each trie level onto a corresponding pipeline stage with its own SRAM memory. In such solutions, the number of pipeline stages depends on the stride used (i.e., the number of bits used to determine which branch to take at each stage—in Fig. 1 and in the rest of this paper, we use 1-bit strides for simplicity). Therefore, the lookup pipeline will require a rather high number of separate SRAMs (up to 33 in the case of IPv4). The FPGA is a natural hardware choice for implementing the SRAM-based pipeline, as it contains hundreds of separate SRAMs inside. Nevertheless, the on-chip SRAM is generally a scarce resource that should be allocated and utilized efficiently or be complemented by external SRAMs [8]. One major issue here is that the shape of a trie determines the size of the SRAM needed in each stage

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE I
ANALYSIS OF REAL ROUTING TABLES

FIB	# of IPv4 prefixes	# of nodes in the trie	# of leaf prefixes	# of nodes in the trimmed trie
rrc00	368057	905941	332409 (90.31%)	110109 (12.15%)
rrc01	358925	880946	325667 (90.73%)	103326 (11.73%)
rrc03	355603	873608	322419 (90.67%)	102984 (11.80%)
rrc04	366656	903163	332962 (90.81%)	104169 (11.53%)
rrc05	358355	879902	324594 (90.58%)	104457 (11.87%)
rrc06	351919	863114	319654 (90.83%)	100819 (11.68%)
rrc07	361881	888468	327781 (90.58%)	106517 (11.99%)
rrc10	355106	871466	321995 (90.68%)	102833 (11.80%)
rrc11	361708	888394	327742 (90.61%)	105552 (11.88%)
rrc12	363761	895781	329377 (90.55%)	106584 (11.90%)
rrc13	363057	894876	328942 (90.60%)	106024 (11.85%)
rrc14	361232	885979	327160 (90.57%)	105475 (11.90%)
rrc15	359326	880902	325154 (90.49%)	104536 (11.87%)
rrc16	366711	903062	331674 (90.45%)	108509 (12.02%)

of the pipeline, so it is difficult to achieve high utilization of SRAMs in all stages. While much work has been devoted to this issue [9]–[11], the fact remains that on-chip SRAMs are still insufficient to accommodate the typically large interdomain FIB (as shown in Table I, about 360 K prefixes to date). For example, it has been reported [11] that Optimized Linear Pipeline (OLP) can support 30 K IPv4 prefixes using 3.456 Mb of on-chip SRAMs. Hence, given a state-of-the-art large Virtex-6 FPGA (e.g., XC6VHX565T) with 32 Mb of on-chip SRAMs, only about 277 K IPv4 prefixes can be stored using OLP. This means that the memory size is still a challenge in the SRAM-based lookup pipeline. Some optimizations [12], [13] that reduce the lookup time complexity and the memory requirement of the trie by multilevel hash tables can be adopted in the SRAM-based lookup pipeline to reduce the number of pipeline stages (i.e., the number of separate SRAMs needed) and the total memory requirement in the pipeline.

When external SRAMs are used for complementing the memory resource of trie-based pipelines, a few large levels are moved into external SRAMs [8]. However, the size of those levels is variable, and controlling the memory distribution among these stages is challenging [8]. Therefore, external SRAMs should be overprovisioned, and memory waste can rarely be avoided. In 2–3 tree-based routers [14], the last few stages of the SRAM-based pipeline are moved to external SRAMs. In these routers, a 2–3 balanced tree is built so that the size of memory needed in level $i+1$ is about twice of that in level i. However, it is impractical to find in the market external SRAMs with exact required sizes. Due to this fact, it is hard to avoid memory waste when using 2–3 tree-based routers, and the memory utilization ratio is usually low.

Route updates are handled in the SRAM-based lookup pipelines through a technique known as write bubbles [15], which essentially pack write messages caused by updates into write packets to be injected into the pipeline. Nevertheless, a single port of the SRAM modules is used in a "half-duplex" way for reading and writing in the past pipelines [15], [16]. This means that write bubbles may lead to disruption to the IP lookup process and possible packet drops. Route updates in the current Internet are known to occur frequently, with peak

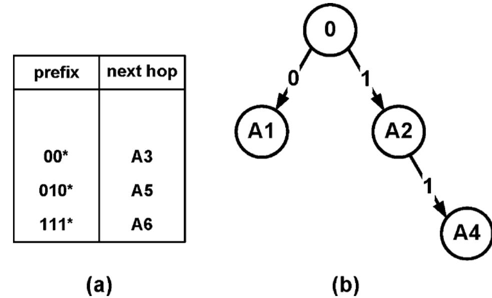

Fig. 2. Two sets after the trie in Fig. 1(b) is partitioned. (a) Corresponding disjoint prefix set and (b) overlapping trie.

update rates affecting thousands of prefixes per second [17]. In the presence of virtual routers, a same network event could trigger simultaneous updates to multiple FIBs, thus increasing the rate of updates to the hardware lookup engine. If write bubbles may lead to packet drops in a single router, the situation is exacerbated drastically when it comes into virtual routers. Fortunately, state-of-the-art FPGAs now integrate dual port SRAMs, capable of concurrent reading and writing. This resolves the problem of disruption caused by updates.

C. Design Motivation and Rationale

In this paper, we aim to achieve fast lookups and fast updates simultaneously by benefiting from the strengths of both the above hardware engines, without being hindered by their weaknesses. The core idea of our solution is based on the empirically observed structure of 1-bit tries built from real FIBs.

1) About 90% of all prefixes are stored in trie leaves [14] and are thus disjoint from each other.
2) When the leaf nodes are removed from the original trie, non-prefix internal nodes that only lead to those leaf nodes can also be removed, and a much smaller trimmed trie remains. The trimmed trie contains, on average, only about 12% of the nodes of the original trie.

The large disjoint prefix set [e.g., see Fig. 2(a)], resulting from property 1), makes a TCAM the ideal component to look these up, as naturally disjoint prefixes do not impose any order constraints within the TCAM, thus making updates trivial: No entry movement is required, and a single write access is sufficient for each update as no prefix expansion is introduced. The small trimmed trie [e.g., see Fig. 2(b)] resulting from the removal of the leaf prefixes from the original trie, which represents the set of prefixes that overlap with the disjoint prefix set thus removed, needs much less memory space and can be stored in the on-chip SRAM-based lookup pipeline in the FPGA. We will refer to this small trimmed trie as "*the overlapping trie*" (we will use the terms *overlapping trie* and *overlapping prefix set* interchangeably throughout the paper). In fact, several such trimmed tries can easily fit in current FPGA's SRAMs. Additionally, by exploiting the dual port capabilities of SRAMs mentioned earlier, updating this SRAM-based pipeline is also trivial.

III. Proposed Architecture

In this section, we describe our hybrid IP lookup architecture with fast updates for a single router. The main ideas of this paper are first to use the above observation to partition a 1-bit trie built

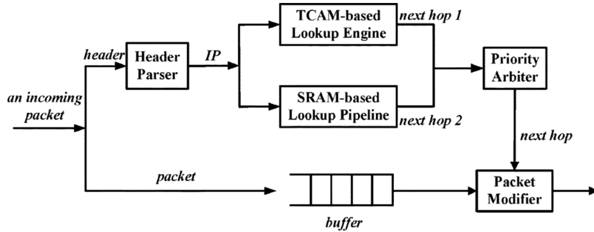

Fig. 3. Hybrid IP lookup architecture.

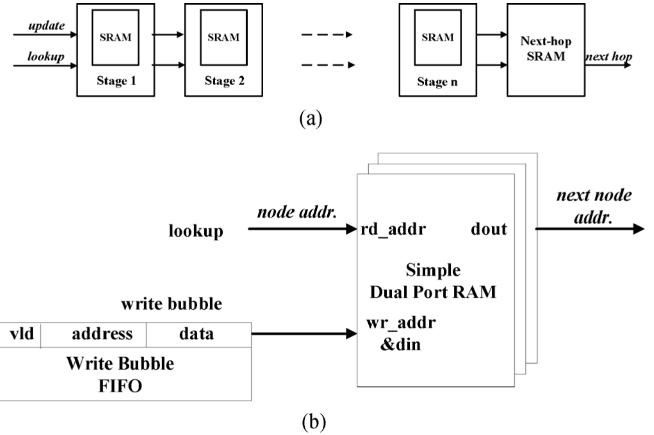

Fig. 4. SRAM-based IP lookup pipeline. (a) n-stage IP lookup pipeline. (b) Single stage of the pipeline.

from the FIB of the router into a large disjoint leaf prefix set and a small trimmed overlapping trie, and then to design a hybrid lookup architecture to accommodate these two sets. The large disjoint leaf prefix set is mapped into an external TCAM-based IP lookup engine, while the small trimmed overlapping trie is mapped into an on-chip SRAM-based IP lookup pipeline in the FPGA.

A. Trie Partitioning Scheme

We use an efficient trie partitioning scheme similar to the set-bounded leaf-pushing algorithm in [14] to partition the 1-bit trie into two prefix sets and to benefit from the observation made in Section II-C. First, all the leaf prefixes in the trie are collected to form a large disjoint prefix set, and thus all the leaf nodes can be removed from the trie. Then, we can further trim the remaining trie by removing nonprefix leaf nodes recursively until all the leaf nodes in the final trimmed trie are prefix nodes. Note that the main difference between our approach and that applied in [14] is that we are not using leaf pushing in the trimmed trie in order to enable fast updates.

Fig. 2 illustrates the results after the trie partitioning scheme is applied to the trie shown in Fig. 1(b). All the leaf prefixes (i.e., prefix 00*, 010*, and 111*) of the original trie [see Fig. 1(b)] are moved to a disjoint prefix set [see Fig. 2(a)], and the corresponding leaf nodes are deleted from the trie. Then, the remaining trie can be further trimmed. For example, the node corresponding to prefix 01* [see Fig. 1(b)] becomes a leaf node, but it does not contain next-hop information (i.e., it has become a non-prefix leaf node), so it can be further removed. The final trimmed trie is shown in Fig. 2(b) and represents the small overlapping prefix set (i.e., a small overlapping trie).

B. Overall Architecture

The hybrid IP lookup architecture is depicted in Fig. 3 and is composed of two IP lookup engines operating in parallel. The large disjoint leaf prefix set [e.g., see Fig. 2(a)] is stored in the TCAM-based lookup engine, while the small overlapping trie [e.g., see Fig. 2(b)] is mapped into the on-chip SRAM-based lookup pipeline. The destination IP address of an incoming packet is extracted in the header parser module and sent to the two lookup engines in parallel. Meanwhile, the packet is stored in a buffer waiting for the next-hop information. After the lookups in both engines are finished (i.e., the lookup in the TCAM-based engine must wait until the lookup in the SRAM-based pipeline is finished), the priority arbiter module manages the priority of the two lookup results and provides the next-hop information that is used to schedule the packet to the corresponding output interface. As the length of the prefix matched in the disjoint prefix set is by design longer than that in the overlapping prefix set, the search result of the TCAM-based lookup engine has a higher priority than that of the SRAM-based lookup pipeline.

C. Optimizations for Fast Updates

Efforts are made in both lookup engines to optimize the update process. To achieve fast updates, only the large disjoint prefix set is stored in the TCAM-based IP lookup engine. In such a disjoint prefix set, a given IP address can only be matched by at most a single prefix. Hence, the prefixes can be arranged in the TCAM without any order constraints, and thus prefixes can be directly inserted in and deleted from the TCAM without entry movements. Moreover, the leaf prefix set is naturally disjoint and no prefix is expanded. Hence, in the worst case, a single write access is enough for any route update. Note that the next-hop information associated with a prefix is stored in an associated SRAM in the TCAM-based lookup engine. The TCAM and its associated SRAM can be written (i.e., updated) independently, which means a prefix and its corresponding next-hop information can be updated concurrently. Therefore, only the time for write accesses to the TCAM is considered.

As explained in Section II, write bubbles may lead to disruption to the IP lookup process in the first generation of SRAM-based pipelines [15], [16], as write and read operations could not be performed simultaneously on the same port of an SRAM. However, current FPGAs, like Xilinx FPGAs, can be configured into a simple dual port (SDP) mode where the SRAM has separate read and write ports [18] that enable simultaneous read and write. Using this mode, we have designed a pipeline with separate lookup and update paths that totally eliminate the disruption [see Fig. 4(a)] as lookups are performed by only accessing the read port, while write bubbles are processed by only accessing the write port.

Update is done by injecting a write bubble into the pipeline. Before injecting, the data to be written into the SRAM of each stage are stored in a write bubble first-in–first-out (FIFO) in the stage [see Fig. 4(b)]. The write bubble visits each stage for one clock cycle. When the *vld* (valid) flag in the top entry of the FIFO is set, it writes the data stored in the FIFO into the SRAM at the corresponding address. Therefore, the write bubble does not need to wait for the data, and it can update each stage in a single clock cycle, achieving the same speed as the lookup.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

As a write bubble and an IP lookup can run at the same speed, and one write bubble is sufficient for a worst-case route update when using the 1-bit trie-based data structure for pipelining [16], an IP lookup never traverses the trie in an inconsistent state in our pipeline. A consistent trie state can be defined as the state before a route update is injected, or after a route update is completed. When IP lookups interleave with route updates, if we can guarantee that the lookup result (i.e., the next-hop information) is always identical to one of those results when the IP lookup is performed on any consistent state, the consistency is said to be maintained, and correct longest-prefix matching can be guaranteed.

We ensure that an IP lookup always traverses the trie in a consistent state in our pipeline as follows.

First, when a lookup is accessing a stage preceding the stage a write bubble is accessing (i.e., the lookup is accessing a stage that is nearer to the entrance of the pipeline), the lookup will be always preceding the write bubble since they go through the pipeline at the same speed, and thus the lookup will always observe the modified nodes caused by the write bubble. Therefore, the lookup result is identical to that of the lookup after the route update is completed.

Second, when a write bubble is accessing a stage preceding the stage a lookup is accessing (i.e., the write bubble is accessing a stage that is nearer to the entrance of the pipeline), the write bubble will never catch up with the lookup, and thus the lookup will always observe the old nodes before modification. Therefore, the lookup result is identical to that of the lookup before the write bubble is injected.

Third, even when a lookup and a write bubble are accessing the same node of the same stage simultaneously, the lookup reads the old node before modification (thanks to the READ_FIRST feature of the SDP SRAM in Xilinx FPGA [18]), and this read–write order is kept when they both move to the next stage. It makes the lookup result always identical to that of the lookup before the write bubble is injected.

We take an IP lookup with IP 011 and the write bubble of inserting $\langle 01^*, A7\rangle$ in Fig. 5(c) to illustrate the consistency. If at some point the IP lookup is visiting node 0, and the write bubble is visiting node A1, the IP lookup result will be A7, which is the same as that after the write bubble is completed. If at some point the IP lookup is visiting node A1, and the write bubble is visiting node 0, the IP lookup result will be A1, which is the same as that before the write bubble is performed. Even if the IP lookup and the write bubble are visiting the same node 0 at some point, the IP lookup always reads the old data in each node, so that the lookup result will be A1, which is the same as that before the write bubble is performed.

In summary, in our proposed architecture, a single write access is sufficient for a worst-case route update in the TCAM-based lookup engine, and route updates have zero impact on the lookup performance in the SRAM-based lookup pipeline.

D. Fast Incremental Updating Algorithms

Fast incremental updating algorithms are desirable in the control plane of the router to translate a route update into updates in the TCAM-based lookup engine and the SRAM-based lookup pipeline. A route update can be classified into three main categories [14]: (1) insertion of a new prefix, (2) deletion of an existing prefix, and (3) modification of an existing prefix (i.e., modifying its next-hop information). The third type of update can easily be performed since it does not change the shape of the trie. However, the first two types are more complex. Insertion of a new prefix or deletion of an existing prefix may affect the partitioning results and lead to prefix changes in the disjoint prefix set and the overlapping trie.

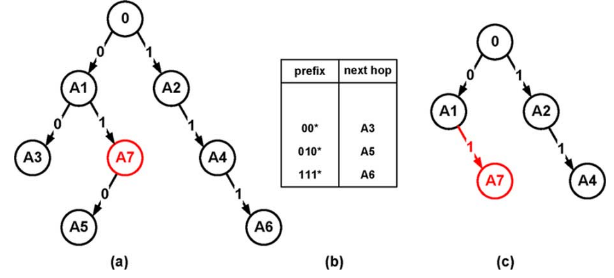

Fig. 5. (a) Insertion of a non-leaf prefix, (b) its corresponding disjoint prefix set, and (c) the overlapping trie.

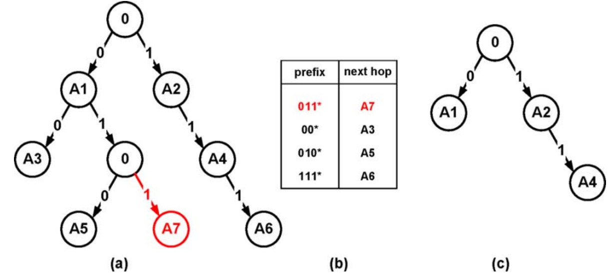

Fig. 6. (a) One example of the insertion of a leaf prefix, (b) its corresponding disjoint prefix set, and (c) the overlapping trie.

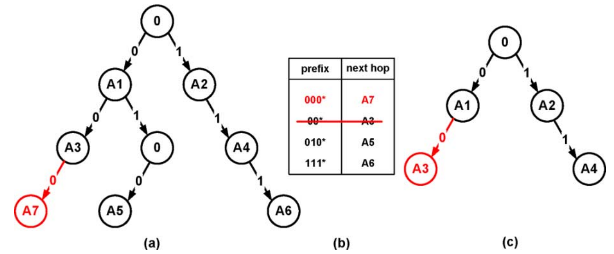

Fig. 7. (a) Another example of the insertion of a leaf prefix, (b) its corresponding disjoint prefix set, and (c) the overlapping trie.

To deal with this, in the control plane of the router, we maintain an auxiliary 1-bit trie built from the FIB. An update operation consists of two phases: in the first phase, the route update is performed on the auxiliary 1-bit trie and changes in the disjoint prefix set and the overlapping trie are found; in the second phase, optimized write accesses are applied to the hybrid hardware lookup architecture.

Insertion of a new prefix in our architecture can be classified into three and only three categories: 1) insertion of a non-leaf prefix; 2) insertion of a leaf prefix, whose nearest ancestor prefix is always a non-leaf prefix before and after the insertion; 3) insertion of a leaf prefix, whose nearest ancestor prefix is turned from a leaf prefix into a non-leaf prefix. The three categories are illustrated in Figs. 5–7, respectively. Without special explanations, the original trie shown in Fig. 1(b) is used as the base of all the update processes in this section.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

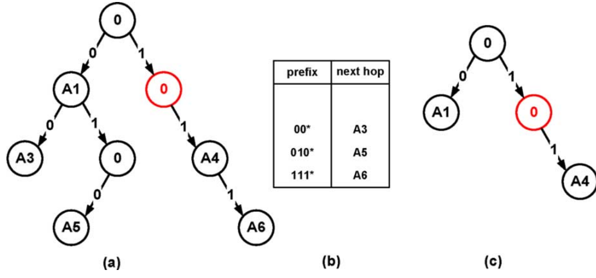

Fig. 8. (a) Deletion of a non-leaf prefix, (b) its corresponding disjoint prefix set, and (c) the overlapping trie.

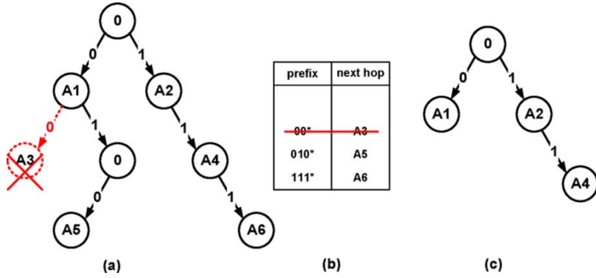

Fig. 9. (a) One example of the deletion of a leaf prefix, (b) its corresponding disjoint prefix set, and (c) the overlapping trie.

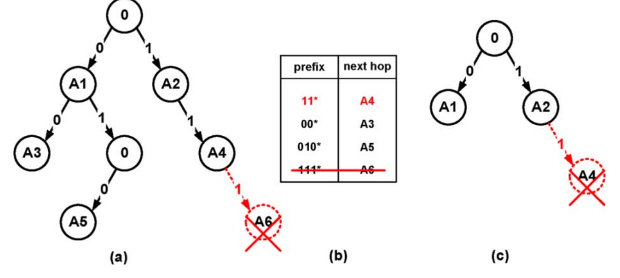

Fig. 10. (a) Another example of the deletion of a leaf prefix, (b) its corresponding disjoint prefix set, and (c) the overlapping trie.

Input: Trie T, and Prefix P which is to be inserted to T.
Output: Changes in the disjoint prefix set S1 and the overlapping prefix set S2.

1 *Insert prefix P into trie T, the new trie is T'*
2 *Find the longest sub-prefix of P in T' : Prefix Q*
3 *if P is a non-leaf prefix in T'*
4 *Add P into S2;*
5 *else if P is a leaf prefix in T'*
6 *if Q is a non-leaf prefix in T*
7 *Add P into S1;*
8 *else if Q is a leaf prefix in T*
9 *Add Q into S2;*
10 *Del Q from S1, and add P into S1;*
11 *end if*
12 *end if*

Fig. 11. Algorithm: Insertion of a prefix.

Fig. 5 shows an example of the insertion of a non-leaf prefix 01*. The insertion of a non-leaf prefix has no impact on the leaf prefixes. Thus, only the new prefix 01* should be inserted into the overlapping trie, and the disjoint prefix set is kept unchanged.

Fig. 6 demonstrates an example of the insertion of a new leaf prefix 011*, whose nearest ancestor prefix (i.e., prefix 0*) is always a non-leaf prefix before and after the insertion. Therefore, this insertion only results in an insertion of prefix 011* in the disjoint prefix set, and the overlapping trie is kept unchanged.

Fig. 7 depicts another example of the insertion of a new leaf prefix 000*, whose nearest ancestor prefix (i.e., prefix 00*) is turned from a leaf prefix (before the insertion) into a non-leaf prefix (after the insertion). This results in three changes in the corresponding disjoint prefix set and overlapping trie: 1) prefix 00* should be inserted into the overlapping trie; 2) prefix 00* should be deleted from the disjoint prefix set; and 3) prefix 000* should be inserted into the disjoint prefix set.

Similar to the insertion, the deletion of an existing prefix in our architecture can also be classified into three and only three categories: 1) deletion of a non-leaf prefix; 2) deletion of a leaf prefix, whose nearest ancestor prefix is always a non-leaf prefix before and after the deletion; 3) deletion of a leaf prefix, whose nearest ancestor prefix is turned from a non-leaf prefix into a leaf prefix. The three categories are illustrated in Figs. 8–10, respectively.

Fig. 8 shows an example of the deletion of an existing non-leaf prefix 1*. The deletion of a non-leaf prefix has no impact on the leaf prefixes, and thus the disjoint leaf prefix set is kept unchanged. Only the deletion of prefix 1* is needed in the overlapping trie.

Fig. 9 demonstrates an example of the deletion of an existing leaf prefix 00*, whose nearest ancestor prefix (i.e., prefix 0*) is always a non-leaf prefix before and after the deletion. Therefore, the deletion of prefix 00* in the original trie only results in a deletion of prefix 00* in the disjoint prefix set, and the overlapping trie is kept unchanged.

Fig. 10 depicts another example of the deletion of an existing leaf prefix 111*, whose nearest ancestor prefix (i.e., 11*) is turned from a non-leaf prefix (before the deletion) into a leaf prefix (after the deletion). It leads to three changes in the corresponding disjoint prefix set and overlapping trie: 1) prefix 111* should be deleted from the disjoint prefix set; 2) prefix 11* should be inserted into the disjoint prefix set; and 3) prefix 11* should be deleted from the overlapping trie.

Note that all scenarios of insertion and deletion are illustrated above, and all of them are easy and fast to be performed. The complete insertion and deletion algorithms are described in Figs. 11 and 12, respectively. Both of these algorithms are performed in software with a time complexity $O(l)$, where l is the length of prefix P to be updated. In both algorithms, one route update generates at most one write operation to each lookup engine. Although it seems that two write operations may be needed in the TCAM in some case, they can actually be combined into just a single write operation. For example, deleting prefix Q and inserting prefix P in the TCAM (see line 10 in Fig. 11) can be combined into one write operation by just overwriting prefix Q with P. Additionally, for one route update, the write order between the two lookup engines should be kept to avoid incorrect longest-prefix matching. For example, the execution of lines 9 and 10 should be kept in the order shown in Fig. 11.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Input: Trie T, and Prefix P which is to be deleted from T.
Output: Changes in the disjoint prefix set S1 and the overlapping prefix set S2.

```
1  Delete prefix P from trie T, the new trimmed trie is T'
2  Find the longest sub-prefix of P in T: Prefix Q
3  if P is a non-leaf prefix in T
4     Del P from S2;
5  else if P is a leaf prefix in T
6     if Q is a non-leaf prefix in T'
7        Del P from S1;
8     else if Q is a leaf prefix in T'
9        Del P from S1, and add Q into S1;
10       Del Q from S2;
11    end if
12 end if
```

Fig. 12. Algorithm: Deletion of a prefix.

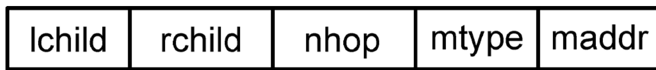

Fig. 13. Node data structure of the auxiliary trie.

Otherwise, prefix Q will disappear in both lookup engines temporarily, which may lead to incorrect longest-prefix matching during updating.

E. Memory Management

Memory management is closely related to the route update process, and the ability to apply an incremental route update makes memory management schemes necessary [16]. Simple and efficient memory management schemes are desirable in order to decrease memory management overhead while updating. The properties of our hybrid architecture make memory management trivial.

In our TCAM-based lookup engine, only the disjoint prefix set is accommodated, and thus entries in the TCAM can be arranged without any order constraints. Management problems in this part are how to keep track of prefix locations and how to manage the allocation and deallocation of the TCAM space.

For the modification or deletion of an existing prefix in the TCAM, the location (i.e., address) of that prefix should be obtained first, and then a write operation can be performed exactly at that location. A search instruction can be issued to the TCAM in order to get the corresponding location. However, the search is an extra TCAM operation, which would add a disruption cycle to the IP lookup process and increase the update overhead. In order to avoid the extra search operation on the TCAM, we keep track of prefix locations in the auxiliary trie in the control plane. The node data structure of the auxiliary trie is shown in Fig. 13. Each node contains five fields, which are the following: left child pointer (*lchild*), right child pointer (*rchild*), next-hop pointer (*nhop*), memory type (*mtype*), and memory address (*maddr*). Here, *mtype* indicates the memory type, with 0 denoting the TCAM, and i $(i > 0)$ denoting the SRAM in the ith stage of the SRAM-based pipeline (note that the auxiliary trie is also used to keep track of node locations in the SRAM-based pipeline). *maddr* denotes the memory address. From the above section, we know that an update operation should first be performed on the auxiliary trie. In this phase, we can get the location of the prefix to be updated in the trie node in the control plane, without the need of an extra search operation on the TCAM.

The insertion of a new prefix requires the allocation of a new entry in the TCAM, and the deletion of an existing prefix incurs the deallocation. In our approach, the unused entries in the TCAM are managed by an auxiliary queue in the control plane, in which a single element stores the address of an unused entry in the TCAM. The allocation of a new entry simply corresponds to a "dequeue" operation of the queue and the deallocation corresponds to an "enqueue" operation. Such a simple and efficient management scheme can work well since the prefixes can be stored in the TCAM with arbitrary orders in our architecture.

In our SRAM-based lookup pipeline, a very small overlapping trie is accommodated. Node locations in the pipeline are also stored in the auxiliary trie. As there are multiple stages and each has its own SRAM, a node location in the pipeline can be represented by a stage ID and a memory address in that stage (i.e., the *mtype* and *maddr* fields in the node data structure of the auxiliary trie). Additionally, the scheme used to allocate and deallocate the SRAM space is similar to that in the TCAM part, but with the difference that each stage of the pipeline is associated with an auxiliary queue in the control plane.

Additionally, specific memory management problems in the SRAM-based pipeline are how to balance the memory across pipeline stages and how it affects the update process.

Many memory balance approaches [9]–[11] have been proposed to date. The OLP approach in [11] achieves an almost perfect balanced distribution, and thus the memory space of on-chip SRAMs can be well utilized. Therefore, the OLP approach in [11] can be used in our SRAM-based pipeline to balance the memory distribution across stages. However, OLP is a static mapping approach, which means that a balanced memory distribution can be achieved only for a given static trie. The effects of incremental updates are not considered in OLP.

The deletion and modification of an existing prefix can be easily performed in the balanced pipeline, as they can be implemented by just rewriting at most one node in each stage, which can be done via a single write bubble. The insertion of a new prefix may trigger the remapping of several new nodes. Mapping new nodes incrementally across pipeline stages affects the memory distribution and a balanced distribution is desirable after incremental mapping. We use the same scheme as in [8] to support incremental mapping. If m nodes caused by an incremental insertion are to be remapped, m stages with the lowest memory utilization are selected (this can be done by comparing the size of each auxiliary queue associated to each stage of the pipeline) to store those nodes. After those stages are selected, an unused entry in each of those stages can be allocated by using the "dequeue" operation on its corresponding auxiliary queue. Note that we use a 1-bit trie with a same node size in the pipeline, hence the allocation and deallocation can not incur any fragmentation. After the entries are allocated, a single write bubble is injected into the pipeline to complete the incremental mapping. Therefore, even if the memory balance approach [11] and

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

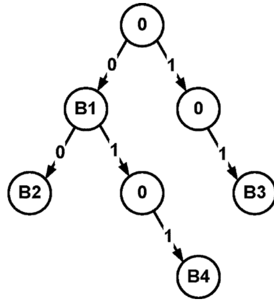

Fig. 14. Another sample trie.

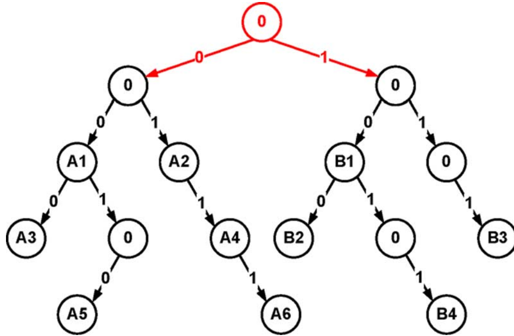

Fig. 15. Merged trie using the virtual prefix technology.

incremental mapping [8] are applied, incremental updates can still be easily performed, and one write bubble is still enough for one route update in our SRAM-based lookup pipeline. Note that OLP and the incremental mapping can cooperate well with each other, and the details are shown in [8].

IV. Lookup for Virtual Routers

We have described the hybrid IP lookup architecture for a single router in previous sections. Nonetheless, our lookup architecture can scale well to support virtual routers naturally.

A virtual router platform contains multiple FIBs. Two common approaches have been proposed so far to merge multiple FIBs together, i.e., virtual prefix technique [14] and trie overlap approach [19]. We will show how these two approaches can be applied respectively in our hybrid architecture to support virtual routers.

A. Virtual Prefix Technique

In this scheme, by appending a unique virtual router ID (VID) before the prefix, we form a virtual prefix. This ensures that the virtual prefix sets belonging to different virtual routers are not overlapping. Hence, the virtual prefix sets of all FIBs can be directly put together to form a large merged FIB. Taking two simple tries as an example, if we assign a VID 0 to the trie shown in Fig. 1(b) and a VID 1 to the trie shown in Fig. 14, they can be combined into a large trie, as shown in Fig. 15.

Using this scheme, the merged trie has the same feature as that in a single router since leaf prefixes in each individual trie are still leaf prefixes in the merged trie, and that is also true for the overlapping trie. Therefore, the trie partitioning scheme applied before is still suitable for the merged trie, and a large disjoint leaf prefix set and a relatively small overlapping trie are generated. Then, the merged disjoint prefix set can be mapped into the

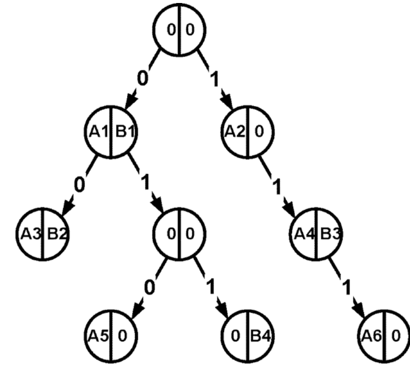

Fig. 16. Merged trie using trie overlap without leaf pushing.

external TCAM-based IP lookup engine, and the merged overlapping trie can be mapped into the on-chip SRAM-based IP lookup pipeline. This makes the architecture depicted in Fig. 3 still suitable for virtual routers with a slight modification. The IP address used to search both lookup engines should be changed to a virtual IP address (VIP) by appending a VID to an IP address. This is performed in the header parser module shown in Fig. 3.

From this point, the update process in virtual routers is similar to that in a single router. When a route update is to be performed on one FIB of virtual routers, the same fast incremental updating algorithm described before is applied on the auxiliary 1-bit trie to find the changes in its disjoint prefix set and overlapping trie, with the difference that now the new prefix to be updated must be constructed by concatenating the prefix with the VID. Taking the insertion in Fig. 7 as an example, and assuming that it is performed in a virtual router instance with a VID 0, the changes in the final merged sets are as follows: 1) virtual prefix 000* should be inserted into the overlapping trie; 2) virtual prefix 000* should be deleted from the disjoint prefix set; and 3) virtual prefix 0000* should be inserted into the disjoint prefix set.

As mentioned before, one route update causes at most one write operation on each lookup engine for a single router. This remains valid for virtual routers; any route update in an FIB of virtual routers needs at most one write operation on each lookup engine.

B. Trie Overlap Approach

Using the virtual prefix technique, the size of the combined trie is equal to the sum of that of individual tries, and it increases linearly as the number of tries increases. In order to reduce the memory consumption and improve the scalability of virtual routers, Jing Fu *et al.* [19] proposed an efficient trie merging approach by sharing nodes among different tries. A common prefix set is built from all the tries to be merged, and then the nodes corresponding to the same prefix in different tries can be merged into one node. Using this scheme, the number of nodes in the merged trie is significantly smaller than the sum of that of individual tries. The trie shown in Fig. 16 is the merged trie after merging the trie shown in Fig. 1(b) and the trie shown in Fig. 14. Note that leaf pushing is not performed after merging.

Whether our hybrid architecture is still suitable for this merging approach depends on the properties of the merged trie. To validate the properties, we have merged fourteen real IPv4

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE II
ANALYSIS OF THE MERGED TRIE

# of tries	# of prefixes	# of nodes	# of leaf prefixes	# of nodes in the trimmed trie
1	368057	905941	332409 (90.31%)	110109 (12.15%)
2	372999	919230	337127 (90.38%)	110770 (12.05%)
3	373535	920501	337600 (90.38%)	110903 (12.05%)
4	382537	946818	346521 (90.59%)	111245 (11.75%)
5	382886	948016	346812 (90.58%)	111415 (11.75%)
6	382887	948017	346813 (90.58%)	111415 (11.75%)
7	388425	962197	351566 (90.51%)	114140 (11.86%)
8	389786	966729	352763 (90.50%)	114685 (11.86%)
9	390384	968938	353230 (90.48%)	114936 (11.86%)
10	393079	978074	355723 (90.50%)	115708 (11.83%)
11	396004	987932	358509 (90.53%)	116226 (11.76%)
12	397768	992125	359976 (90.50%)	116830 (11.78%)
13	401651	1000352	362875 (90.35%)	118500 (11.85%)
14	401750	1000769	362950 (90.34%)	118578 (11.85%)

routing tables (see Table I) using this approach, and then the trie partitioning scheme is applied. Fortunately, we observe that the merged trie still has the same properties as that in a single router, i.e., about 90% of prefixes are leaf prefixes, and the node of the trimmed trie is about 12% of that of the original trie (see Table II for more details). These properties form the base of our hybrid architecture. Therefore, after merging multiple FIBs using trie overlap without leaf pushing, our approach can still be used for achieving fast lookups and fast updates. The proposed update algorithms and hybrid architecture can also be changed slightly to suit this merging approach.

V. Performance Evaluation

A. Analysis of Real Routing Tables

Fourteen real IPv4 core routing tables have been collected from RIPE RIS Project [20] on May 20, 2011. Analysis is performed on these real routing tables to validate the advantage of the trie partitioning scheme. The analysis results of each individual routing table are shown in Table I.

The number of IPv4 prefixes and leaf prefixes in each FIB are shown respectively in column *# of IPv4 prefixes* and *# of leaf prefixes*. We can see that for all the 14 FIBs, more than 90% of the prefixes are leaf prefixes. This is expected since most of IP address prefixes are around 24 bits long, and most of them are disjoint leaf prefixes. The number of nodes in the original trie is represented in column *# of nodes in the trie*. We apply the trie partitioning scheme on these 14 FIBs, respectively. After moving the leaf prefixes into a disjoint leaf prefix set and trimming the trie further, the number of nodes remaining in the final trimmed trie is shown in column *# of nodes in the trimmed trie*. The results show that after trimming, the number of remaining nodes is about 12% of that of the original trie. These observations confirm the initial empirical finding that is the base of the trie partitioning scheme.

Based on the above analysis, the following conclusions can be drawn.

1) Using the partitioning scheme, most of the prefixes are moved to external TCAMs. Meanwhile, all of them are naturally disjoint, and they can be stored without any order constraints. This feature can be used to guarantee fast updates in a TCAM.
2) After removing the leaf nodes, the amount of memory needed in the SRAM-based pipeline is reduced significantly. Hence, the memory size issue of on-chip SRAM-based pipelines in the FPGA can be well addressed.

The above conclusions still hold for virtual routers as each router will have an FIB that will validate the above properties. When using the virtual prefix technology for merging (e.g., Fig. 15), the leaf prefixes of each individual trie are still the leaf prefixes in the merged trie, and that is also true for the overlapping tries. Therefore, the properties of the merged trie are kept exactly as that in each individual trie. When using the trie overlap approach without leaf pushing for merging (e.g., Fig. 16), the leaf prefix set and the overlapping trie in the merged trie are not exactly as that in individual tries, as some leaf prefixes in individual tries turn into non-leaf prefixes in the merged trie. In order to validate whether the trie partitioning scheme is still valid for the merged trie when using trie overlap, we merge a number of tries and then partition the merged trie. The analysis results are shown in Table II. The results show that, as we increase the number of tries, the number of leaf prefixes in the merged trie is always about 90% of that of the total prefixes (see column *# of leaf prefixes* in Table II). Meanwhile, the number of nodes of the trimmed trie is always about 12% of that of the original merged trie (see column *# of nodes in the trimmed trie* in Table II). From Table II, we can observe that, with the increase of the number of tries, the properties of the merged trie are always kept the same as those in a single trie when using the trie overlap approach for scalable virtual routers. Therefore, our hybrid architecture is also a promising candidate for building scalable virtual routers.

B. Throughput Evaluation

We have implemented the proposed hybrid architecture on the PEARL [2] hardware platform we have built previously. PEARL is equipped with a Xilinx Virtex-5 XC5VLX110T-1 FPGA and an IDT IDT75K72100 TCAM. After post place and route, the FPGA achieves a maximum clock frequency of 297 MHz resulting into 297 MLPS for the SRAM-based lookup pipeline. The TCAM has a theoretical maximum throughput of 250 MLPS. Hence, the current PEARL hardware implementation enables a maximum IP lookup throughput of 250 MLPS, or a throughput of 128 Gb/s with 64-B packets, that exceeds largely the throughput requirement of 100G Ethernet. Obviously, if newer and faster FPGAs and TCAMs are used, the performance may be even higher. However, the PEARL platform we used has only 4 Gigabit Ethernet (GbE) interfaces, which allow a maximum input traffic rate of 4 Gb/s. We generated 4 Gb/s traffic of 64–1518-B packets by Spirent TestCenter [21] and connected it directly to our PEARL platform via 4 GbE links. We show in Fig. 17 the measured and theoretical throughput obtained over the PEARL platform with the proposed hybrid IP lookup architecture. The measured maximum throughput is 5.95 million packets per second (MPPS) for 64-B packets, which equals the theoretical maximum packet rate of 4 GbE links.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

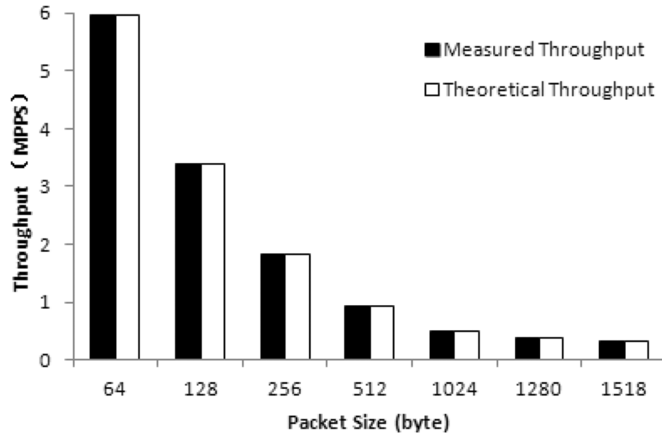

Fig. 17. Throughput of the hybrid architecture.

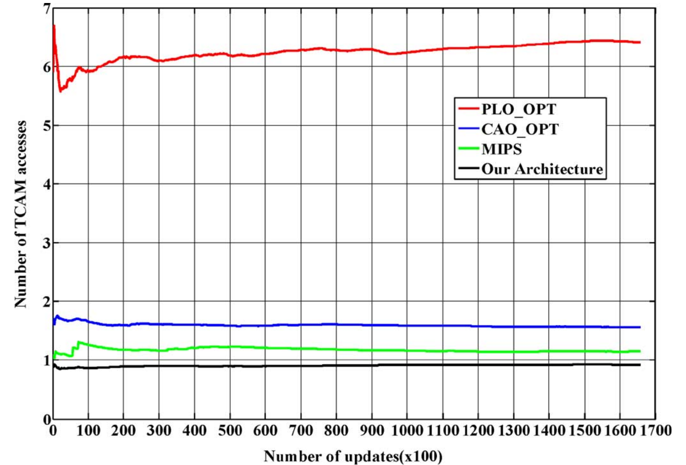

Fig. 18. Running average of the number of TCAM accesses per update on rrc00 routing table.

Note that it is hard to make a fair comparison to throughput measured in other work since the device types and optimization parameters of implementation tools are very different. However, the throughput of our implementation is clearly adequate for practical virtual routers.

C. Update Overhead

The number of TCAM write accesses per update is used as the metric to estimate the update overhead of TCAM-based engines. For the SRAM-based pipeline, we use the number of disrupted lookup cycles per write bubble as the metric of comparison. We have chosen PLO_OPT/CAO_OPT [4], MIPS [5], and write bubbles in [15] and [16] as a comparison basis.

Theoretical Comparison: In the best case, only one TCAM write access is required for each route update in both PLO_OPT and CAO_OPT, and zero TCAM write access is required for each update in both MIPS and our architecture. However, the results in the worst case are quite different. In PLO_OPT, the prefix-length order should be kept, and the empty space is arranged in the center of a TCAM. Therefore, a route update requires at most $W/2$ write accesses to the TCAM, where W is the maximum length of the prefixes (32 for IPv4). In CAO_OPT, the chain-ancestor order should be kept and the empty space is still arranged in the center. Therefore, a route update requires at most $D/2$ write accesses to the TCAM, where D is the maximum length of the chain. Theoretically, D may be up to W. MIPS utilizes leaf pushing to convert the prefix set into an independent (disjoint) prefix set. However, leaf pushing may expand a prefix many times. In the theoretical worst case, a prefix could be expanded to 2^{W-1} prefixes. Therefore, the maximum number of TCAM accesses for one route update is 2^{W-1}. In our hybrid architecture, the prefix set stored in the TCAM is naturally disjoint, and prefix expansion can be totally avoided, and thus one route update leads to at most one write access to the TCAM in any case. The theoretical comparison of the number of TCAM write accesses per update between different schemes is summarized in Table III.

Empirical Comparison: We get from the RIPE RIS project [20] one of the publicly available routing tables rrc00 (see Table I) and 1-h update traces on it. The update traces contain 165 721 updates. Fig. 18 shows the running average of the number of TCAM accesses per update required for all the four compared TCAM update schemes as a function of the number of updates. The average in our proposed hybrid architecture remains persistently under one TCAM access (about 0.91) per update. This is expected since only one TCAM access is required for a leaf prefix update, and zero TCAM access for a non-leaf prefix update. It can be seen that the average number of TCAM accesses in the hybrid scheme is much lower than that of all other three competing solutions. More importantly, the maximum number of TCAM accesses per update, which directly affects the size of the packet buffer required in a lookup engine to avoid packet drops during updating, is precisely equal to one and significantly lower than that of competitor schemes (see Table IV).

TABLE III
THEORETICAL COMPARISON OF THE NUMBER OF TCAM WRITE ACCESSES PER UPDATE

TCAM-based Engines	Maximum	Minimum
PLO_OPT	W/2 (16)	1
CAO_OPT	D/2 (16)	1
MIPS	2^{W-1} (2^{31})	0
Our Architecture	1	0

TABLE IV
COMPARISON RESULTS ON rcc00 ROUTING TABLE

TCAM-based Engines	Maximum	Average	Minimum
PLO_OPT	16	6.42	1
CAO_OPT	4	1.55	1
MIPS	247	1.15	0
Our Architecture	1	0.91	0

Obviously, the number of TCAM accesses per update in our proposed architecture can be proved to be optimal as at most a single write access per update to the TCAM is mandatory. This means that we can guarantee a minimum worst-case update overhead in the TCAM-based lookup engine. The summary of comparison results on routing table rrc00 is shown in Table IV. Last but not least, even if MIPS is able to achieve a performance relative to an average number of write accesses per update that

TABLE V
PACKET DROPS ON rcc00 ROUTING TABLE IN 1 h

TCAM-based Engines	Per Update			Total
	Maximum	Average	Minimum	
PLO_OPT	6.4	2.57	0.4	425465
CAO_OPT	1.6	0.62	0.4	102935
MIPS	98.8	0.46	0	76368
Our Architecture	0.4	0.36	0	60564

is close to 1, the worst-case overhead for a single update is very high (see Tables III and IV).

In [15] and [16], write bubbles are used for route updates in SRAM-based pipelines. Each write bubble may disrupt the IP lookup process for one cycle in the worst case, and minimizing the number of write bubbles reduces the update overhead. In our approach, we have addressed this challenge by devising a pipeline with separate lookup and update paths in order to totally eliminate the disruption to IP lookup process caused by write bubbles.

Note that the whole update process also includes the update time in the auxiliary 1-bit trie in software. This part does not affect the lookup performance in hardware and thus is less important. We have mentioned in Section III-D that the time complexity of updates in software is $O(l)$, where l is the length of prefix P to be updated. Let us consider the worst case, i.e., each IPv4 update will visit 33 trie nodes in software, and each node access needs a memory reference consuming about 60 ns if cache misses. In this case, each update takes at most 1980 ns in the trie, and thus more than 505 051 updates can be preprocessed per second in software, which exceeds largely the peak update frequency of real routers [17].

In summary, each route update leads to at most one write access in the TCAM-based IP lookup engine and has zero impact on the SRAM-based pipeline. Therefore, the update overhead is significantly lower than that of previous work [4]–[16].

D. Packet Drops

Route updates may lead to disruption to IP lookup process and result in packet drops. In this section, we will evaluate this adverse impact. However, it is difficult to evaluate exactly the number of packet drops caused by route updates, as packet drops are also affected by several other factors (e.g., the burst features of network traffic and the size of packet buffers in the router). Nonetheless, we can approximately evaluate it based on some ideal assumptions similar to that in [3]. Let us consider a link rate of 10 Gb/s (i.e., 51.2 ns per packet time for 64-B packets) and a TCAM running at a clock rate of 100 MHz (i.e., 10 ns per clock cycle). A write access to the TCAM actually requires two accesses: loading the rule and loading its corresponding mask, which translates into $10\times 2/51.2 \approx 0.4$ packet drops in the theoretical worst case (no packet buffer). Note that the write access to the next-hop information in the associated SRAM is ignored since it can be performed in parallel with the write access of the TCAM.

Based on the above assumptions, the number of packet drops caused by route updates can be evaluated approximately. Table V shows the number of packet drops caused by route updates on the rrc00 routing table in 1 h (we use the 1-h update traces mentioned in Section V-C).

The number of packet drops per update is shown in column *Per Update*, and the total number of packet drops in 1 h is shown in column *Total*. When compared to the other three solutions, our architecture achieves a smallest number of packet drops per update. This is expected, as we can achieve the lowest update overhead. Moreover, as the number of updates increases, we can achieve a more significant reduction in packet drops (see column *Total* in Table V).

Although the above evaluation is based on some ideal assumptions and the number of packet drops in real routers will be much smaller due to packet buffering, the comparison in Table V is still meaningful. It means that the adverse impact of route updates on packet forwarding can be significantly reduced when using our architecture, which makes it a more promising candidate to reduce packet drops or even to avoid packet drops during updating for real routers.

E. Memory Requirements

The 14 routing tables shown in Table I are collected from core routers, and each has about 360 K prefixes. To show the amount of memory needed in our architecture for storing such a large core routing table, we take the routing table rrc00 (see Table I) as an example.

In the TCAM-based lookup engine, 332 409 leaf prefixes in this routing table should be stored, and thus at least 332 409 entries with a 32-bit size are needed for the TCAM. Therefore, the size of the TCAM should be larger than 10.6 Mb, which is not a big problem for current TCAM devices. The memory capacity of a modern TCAM device with 512 K 40 b entries [22] is sufficient for such a large routing table in our architecture.

In the SRAM-based lookup pipeline, the overlapping trie with 110 109 nodes should be accommodated. To calculate the node size, we use 17 bits to represent the left or right child pointer (a 17-bit pointer can represent at most 131 072 nodes, which exceeds largely the number of nodes of the trimmed overlapping trie), and 8 bits to represent the next-hop pointer. Thus, the node size is 42 bits. Then, the size of on-chip SRAMs needed is only about 4.6 Mb. Nowadays, the capacity of on-chip SRAMs in Xilinx Virtex-6 series FPGA varies from 5.6 to 38.3 Mb [18], and thus all devices in this series are sufficient for the memory requirement of this small overlapping trie.

Note that during memory evaluation, we do not do any memory compaction, and the amount of memory required is a maximum value. Even so, modern devices can still easily satisfy the maximum requirements. Therefore, the memory consumption in our hybrid architecture is reasonable.

F. Memory Utilization

Due to the limited number of available I/O pins in the FPGA or other processing chips (e.g., CPU, Network Processor), only a few external memories can be used in a practical router system. Therefore, the utilization ratio of external memories becomes very important.

In previous solutions [4], because of order constraints, a few empty memory locations (i.e., memory holes) may be kept at all

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

K (i.e., the number of unique prefix lengths) nonempty memory locations in the TCAM in order to further reduce memory movements, which leads to waste of precious TCAM space. In our proposed architecture, the disjoint prefix set can be stored in external TCAMs without any order constraints. As a result, a disjoint prefix set can be mapped into a TCAM until it becomes full. Moreover, multiple external TCAMs can be cascaded to store more prefixes. This means that all TCAMs except the last one, where we should spare some empty space for further updating, can attain a memory utilization ratio of 100%. Therefore, memory waste can be avoided.

Additionally, the memory utilization among on-chip SRAMs in the FPGA can also be well balanced using the scheme proposed in [11].

VI. Discussions

A. IPv6 Case

In the previous sections, we use IPv4 FIBs to describe the proposed approach. In fact, our approach can scale well to support IPv6. We extracted an IPv6 FIB from router rrc00 in Table I and performed the trie partitioning scheme on its corresponding trie. In this IPv6 trie, about 92% of the prefixes are leaf prefixes, and only 9% of nodes are left in the trimming trie. These results are consistent with the observations in the case of IPv4, which are the basis of the proposed hybrid architecture. Therefore, if the hybrid architecture is used in IPv6, the same benefits can be achieved as that in IPv4. However, the length of the IPv6 address is much longer. We need a TCAM with larger entry size for IPv6 prefixes, and an FPGA containing more than 129 separate on-chip SRAMs to implement the IPv6 trie pipeline. Current NetLogic TCAMs can be configured to have 160-bit entries [22], and many Xilinx FPGAs contain more than 200 on-chip SRAMs inside [18]. Both of these chips can be used in the hybrid architecture to support IPv6.

B. Memory Footprint

Although external TCAMs can be fully utilized and only 90% of the prefixes of the FIBs are stored in TCAMs, achieving a smaller memory footprint in a TCAM is desirable. For example, an existing large TCAM can accommodate up to 1024 K 40-bit entries [22]. However, there are more than 300 K leaf prefixes in a single FIB (see Table I), which means that only leaf prefixes of three virtual router FIBs can be accommodated in this TCAM. Indeed this memory scalability issue exists for all TCAM-based solutions, as well as for all SRAM-based pipelines.

In our approach, before partitioning the trie, the pruning [23] and the compression [5] schemes can be used to remove the redundancies in the trie, and thus the size of both prefix sets and the memory requirement in both the TCAM engine and the SRAM pipeline can be reduced. On the other hand, after partitioning, merging can be performed in each part to reduce the memory requirement for multiple FIBs. The common leaf prefixes among different FIBs can be shared [24] to reduce the TCAM memory requirement for multiple disjoint leaf prefix sets, and node sharing among different tries [19], [25] can be performed to reduce the memory requirement of on-chip SRAMs for multiple overlapping tries. However, it is noteworthy that there is a tradeoff between memory footprint and update overhead, i.e., compressing too much the data structure may drastically increases the update overhead. This tradeoff should be considered during compacting in practice.

VII. Concluding Remarks

In this paper, we mainly focus on the route update challenge for high-speed routers. An efficient trie partitioning scheme motivated by the observation that more than 90% of prefixes in the 1-bit trie are naturally disjoint leaf prefixes is applied to convert a 1-bit trie into a large disjoint leaf prefix set and a small overlapping trie. The leaf prefix set is naturally disjoint and can be easily mapped into an external TCAM-based lookup engine, thus avoiding entry movements and prefix expansion, and enabling a single write access for each update of a leaf prefix. Additionally, the simplified memory management of TCAMs results in a utilization ratio of TCAMs close to 100%. After removing the leaf nodes, the remaining trie can be further trimmed, resulting in an overlapping trie that contains only about 12% of the nodes of the original trie. The overlapping trie is thereafter implemented in an SRAM-based lookup pipeline with significantly lower memory requirement. In the context of virtual routers, multiple such overlapping tries can be accommodated in the on-chip SRAMs of existing FPGAs. Moreover, by exploiting the dual-port SRAMs in Xilinx FPGA, we design an SRAM-based pipeline with separate lookup and update paths. The pipeline enables simultaneous update and lookup operations without any collision. Therefore, route updates have zero impact on our dual-path SRAM-based pipeline.

The fast incremental updating algorithms we implemented guarantee that, in any case, any route update leads to at most one write access in our TCAM-based lookup engine, and at most one write bubble in our SRAM-based lookup pipeline (we can ignore the update overhead in our SRAM-based lookup pipeline since updates have zero impact on lookups). Therefore, we only need to lock the TCAMs for the time of at most one write access during each update. This update overhead is significantly lower than that of previous work.

In the context of virtual routers, two common FIB merging approaches can be applied in conjunction with our hybrid architecture in order to build scalable virtual routers.

The performance evaluation shows that the throughput is sufficient for 100G Ethernet routers, the update overhead is significantly lower than that of previous work, and the utilization ratio of most external high-capacity memories can be up to 100%. While the memory consumption of our proposed scheme is reasonable, we will study, as future work, compact data structures that can be applied to reduce memory consumption in both engines, while retaining the fast update property of the architecture.

References

[1] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup," in *Proc. IEEE INFOCOM*, 2008, pp. 2458–2466.

[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 Commun. Mag.*, vol. 49, no. 7, pp. 71–77, 2011.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[3] Z. J. Wang, H. Che, M. Kumar, and S. K. Das, "CoPTUA: Consistent policy table update algorithm for TCAM without locking," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1602–1614, Dec. 2004.
[4] D. Shah and P. Gupta, "Fast updating algorithms for TCAM," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, Jan.–Feb. 2001.
[5] G. Wang and N. F. Tzeng, "TCAM-based forwarding engine with minimum independent prefix set (MIPS) for fast updating," in *Proc. IEEE ICC*, 2006, pp. 103–109.
[6] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, Feb. 1999.
[7] S. Sikka and G. Varghese, "Memory-efficient state lookups with fast updates," *Comput. Commun. Rev.*, vol. 30, no. 4, pp. 335–347, 2000.
[8] W. Jiang and V. K. Prasanna, "Towards practical architectures for SRAM-based pipelined lookup engines," in *Proc. IEEE INFOCOM Work-in-Progress Track*, 2010, pp. 1–5.
[9] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in *Proc. ICSA*, 2005, pp. 123–133.
[10] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "CAMP: Fast and efficient IP lookup architecture," in *Proc. ACM/IEEE ANCS*, 2006, pp. 51–60.
[11] W. Jiang and V. K. Prasanna, "A memory-balanced linear pipeline architecture for trie-based IP lookup," in *Proc. IEEE HOTI*, 2007, pp. 83–90.
[12] N. Futamura, R. Sangireddy, S. Aluru, and A. K. Somani, "Scalable, memory efficient, high-speed lookup and update algorithms for IP routing," in *Proc. ICCCN*, 2003, pp. 257–263.
[13] R. Sangireddy, N. Futamura, S. Aluru, and A. Somani, "Scalable, memory efficient, high-speed IP lookup algorithms," *IEEE/ACM Trans. Netw.*, vol. 13, no. 4, pp. 802–812, Aug. 2005.
[14] H. Le, T. Ganegedara, and V. K. Prasanna, "Memory-efficient and scalable virtual routers using FPGA," in *Proc. ACM/SIGDA FPGA*, 2011, pp. 257–266.
[15] A. Basu and G. Narlikar, "Fast incremental updates for pipelined forwarding engines," *IEEE/ACM Trans. Netw.*, vol. 13, no. 3, pp. 690–703, Jun. 2005.
[16] J. Hasan and T. N. Vijaykumar, "Dynamic pipelining: Making IP-lookup truly scalable," in *Proc. ACM SIGCOMM*, 2005, pp. 205–216.
[17] "The BGP instability report," [Online]. Available: http://bgpupdates.potaroo.net/instability/bgpupd.html
[18] Xilinx, San Jose, CA, USA, "Xilinx FPGA," [Online]. Available: http://www.xilinx.com/
[19] J. Fu and J. Rexford, "Efficient IP-address lookup with a shared forwarding table for multiple virtual routers," in *Proc. ACM CoNEXT*, 2008, pp. 21:1–21:12.
[20] RIPE NCC, Amsterdam, The Netherlands, "RIPE RIS raw data," 2011 [Online]. Available: http://www.ripe.net/data-tools/stats/ris/ris-raw-data
[21] Spirent, Crawley, U.K., "Spirent TestCenter," [Online]. Available: http://www.spirent.com/Solutions-Directory/Spirent-TestCenter/
[22] NetLogic, New York, NY, USA, "NL9000 RA knowledge-based processors," 2009.
[23] H. Liu, "Routing table compaction in ternary CAM," *IEEE Micro*, vol. 22, no. 1, pp. 58–64, Jan.–Feb. 2002.
[24] L. Luo, G. Xie, S. Uhlig, L. Mathy, K. Salamatian, and Y. Xie, "Towards TCAM-based scalable virtual routers," in *Proc. ACM CoNEXT*, 2012, pp. 73–84.
[25] H. Y. Song, M. Kodialam, F. Hao, and T. V. Lakshman, "Building scalable virtual routers with trie braiding," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

Layong Luo received the B.S. degree in electronic science and technology from the University of Science and Technology of China, Hefei, China, in 2004, and is currently pursuing the Ph.D. degree at the Chinese Academy of Sciences (CAS), Beijing, China.

His research interests include programmable virtual routers and high-performance packet lookup algorithms.

Gaogang Xie (M'12) received the Ph.D. degree in computer science from Hunan University, Changsha, China, in 2002.

He is currently a Professor and Director of Network Technology Research Center with the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China. His research interests include programmable virtual routers, future Internet architecture, and Internet measurement.

Yingke Xie received the Ph.D. degree in computer science from the Chinese Academy of Sciences (CAS), Beijing, China, in 2000.

He is an Associate Professor with the Institute of Computing Technology (ICT), CAS. His research interests include programmable virtual routers, reconfigurable computing, and network system architecture.

Laurent Mathy (M'93) received the Ph.D. degree in computer science from Lancaster University, Lancaster, England, in 2000.

He is a Professor with the Electrical Engineering and Computer Science Department, University of Liège, Liège, Belgium. He is also a Visiting Professor with the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China. Prior to joining the University of Liège, he was a Professor with Lancaster University. His research interests include protocol design, Internet architecture, and design and optimization of networked systems.

Kavé Salamatian received the M.B.A. degree from Isfahan University of Technology, Isfahan, Iran, in 1993, and the Ph.D. degree in computer science from Paris SUD-Orsay University, Orsay, France, in 1998.

He is a Full Professor with the University of Savoie, Annecy-le-Vieux, France. He was previously a Reader with Lancaster University, Lancaster, U.K., and an Associate Professor with the University Pierre et Marie Curie, Paris, France. He also worked on the market floor as a Risk Analyst and enjoyed being an Uurban Traffic Modeler for some years. During his Ph.D. studies, he worked on joint source channel coding applied to multimedia transmission over Internet. His main areas of research are Internet measurement and modeling and networking information theory. He is working these days on figuring out if networking is a science or just a hobby, and if it is a science, what are its fundamentals.