Two-Level Tries: A General Acceleration Structure for Parallel Routing Table Accesses

Lai Mingche and Gao Lei

Abstract: **The stringent performance requirement for the high efficiency of routing protocols on the Internet can be satisfied by exploiting the threaded border gateway protocol (TBGP) on multi-cores, but the state-of-the-art TBGP performance is restricted by a mass of contentions when racing to access the routing table. To this end, the highly-efficient parallel access approach appears to be a promising solution to achieve ultra-high route processing speed. This study proposes a general routing table structure consisting of two-level tries for fast parallel access, and it presents a heuristic-based divide-and-recombine algorithm to solve a mass of contentions, thereby accelerating the parallel route updates of multi-threading and boosting the TBGP performance. As a projected TBGP, this study also modifies the table operations such as insert and lookup, and validates their correctness according to the behaviors of the traditional routing table. Our evaluations on a dual quad-core Xeon server show that the parallel access contentions decrease sharply by 92.5% versus the traditional routing table, and the maximal update time of a thread is reduced by 56.8% on average with little overhead. The convergence time of update messages are improved by 49.7%.**

Index Terms: **Multi-cores, two-level tries, route update.**

I. INTRODUCTION

With the tremendous growth in the scale of the global Internet, the border gateway protocol (BGP) [1] is becoming intricate due to its complex design and a variety of protocol extensions; this requires increasing central processing unit (CPU) resources on the control planes of routers. The stringent requirement for the highly-efficient BGP and the substantial increase in the applications on the Internet have produced many challenges [2]–[5]. Agarwal *et al.* [6] examined the BGP data from 196 Cisco routers in the Sprint Internet protocol (IP) network, and the analysis results revealed that BGP processes contributed over 60% of the CPU load. This fast consumption of computing resources is attributed to the routing dynamics [7]. The increasing connection density of autonomous system (AS) has also introduced a great demand to satisfy the performance requirements by the incremental peer sessions. Black-box tests [8] on commercial routers showed that they could not afford to maintain the BGP peer sessions, when the number of sessions went up to 250. In this context, the pass-through time was increased by several orders, and thus BGP could not provide sufficient efficiency to guarantee the quality of delay- and loss-sensitive applications. In addition, some other instability events, like BGP outburst update [9], route oscillations [10]–[11], slow convergence [12]–[13], configuration errors [14], and malicious attacks [14], further deteriorate BGP efficiency and present a higher BGP performance demand.

Many technologies [15]–[22] based on distributed processing and parallel computing have been put forward to improve the performance of transport and routing protocols. Bjorkman *et al.* [15] analyzed the parallelism of transmission control protocol/Internet protocol (TCP/IP) on the message, connection, protocol, and task levels, and they proposed a queuing network model to predict the performance when capturing the effects of lock and memory contentions. Xiao *et al.* [16] presented a parallel routing table computation approach for open shortest path first (OSPF), which divided the OSPF area into several regions and calculated the routing table by different nodes in parallel. Klockar *et al.* [17] proposed a distributed router based on a modularized BGP, where multiple BGP processes worked in parallel and each node selected a path according to its local route information, and the forwarding table was divided into several parts and updated by a BGP proxy. Since it did not hold a consistent view of the global routing table on each node, there would be multiple forwarding paths for the same destination in the forwarding table. Zhang *et al.* [18] proposed a fully-distributed BGP model by distributing route computation to multiple agents. It extended BGP and put forward clustered BGP (CBGP) for the synchronizations of routes and status information. A tree-based distributed model of BGP was devised in [19], which gave the corresponding algorithms for dispatching subtasks under two typical scalable router systems. Xu *et al.* [20] then considered the task balance on different nodes based on two kinds of iteration trees, decreasing the communication cost among nodes and increasing the computing efficiency by 91.8% compared to neighbor-based division schemes. The above studies have enhanced the BGP performance effectively, but they are constrained to apply to clusters or multi-processors, where the long communication latency and high synchronization overhead of distributed memory architecture always bottleneck further BGP performance improvements. The work in [21] tried to exploit the speculative parallelism in BGP, but the high complexity of verification and a great deal of overhead for lock operations limited its speedups.

The ubiquity of large-scale multi-core processors has radically changed the landscape of parallel processing. The multi-core integrates multiple computing entities, favoring the use of thread-level parallelism. Meanwhile, it adopts shared memory to ease the communication bottleneck in clusters or multi-processors, and it provides faster synchronization operations as well as higher communication bandwidth, giving us an incentive to explore multi-threading in BGP. In this study, an orig-

Manuscript received February 08, 2009; approved for publication by Hussein Mouftah, Division III Editor, May 27 , 2011.

This work was supported by NSFC 60903039 and 61103188.

The authors are with the Department of Computer, National University of Defense Technology, Changsha, China, email: {mingchelai,gaolei}@nudt.edu.cn.

inal threaded border gateway protocol (TBGP) is developed on a multi-core system. It empowers multiple threads processing in parallel with decoupled executions among different peer sessions, thereby efficiently improving BGP performance. But evaluation revealed that the route update phase, which consumes the most resources, also acts as the critical portion of the BGP process. The TBGP scheme encountered a mass of contentions when multiple threads raced to access the shared routing table, which significantly bottlenecked the overall performance. To boost the TBGP performance on multi-core processors, optimizing parallel access to the routing table seems to be a critical problem. The state-of-the-art on parallel access to the routing table is still limited. Liang *et al.* [23] proposed a parallel IP routing lookup system, which partitioned the routing table into non-overlapping prefix sets by different heights in trie, and achieved fast IP lookup. In [24], the CREW-based IP lookup was introduced to simultaneously search the incoming IP addresses on multiple processors by sorting prefixes based on range. The author of [25] proposed a parallel lookup scheme by evenly allocating route entries on ternary content addressable memory (TCAM) chips and balancing the traffic load to maximize the lookup throughput. The algorithms above, however, mainly resolve fast prefix lookup on the forwarding plane, while the emphasis of this paper lies on the fast route update of multi-threaded BGP on the control plane, providing a method to accelerate concurrent route update.

In this study, we report a general parallel table access approach for TBGP to release the routing table access contentions and accelerate the route update process. The main contributions of our study are concentrated on three aspects. First, the two-level tries structure and its parallel access approach are proposed to satisfy the route update parallelization requirements of multiple threads, efficiently breaking the restriction of sequential table accesses and eliminating the performance bottleneck of routing updates in the original TBGP. Second, a divide-and-recombine algorithm that incurs little computational complexity is presented to significantly reduce the number of parallel access contentions by balancing the accesses between different subtrie sets, thereby further accelerating the parallel route updating of the multi-threading and boosting the TBGP performance. Third, two-level tries operations that remain consistent with the behaviors of traditional routing tables are also provided to achieve a higher rate of route updating.

II. TBGP ARCHITECTURE

A router connected to multiple peer neighbors is always exchanging information through BGP, which processes the decoupled characteristic [1] that the messages from different neighbors can be processed in parallel due to their weaker correlations, while those from the same neighbor have to be processed in order due to their close correlations. Thus, as the routers shift to multi-core systems, we propose the TBGP protocol to exploit the potential parallelism by dispatching multiple neighbor sessions on different parallelized threads, thereby improving the protocol efficiency. In general, TBGP is composed of one master thread and multiple slave threads, as shown in Fig. 1. The master thread has the responsibility of initializing the process,

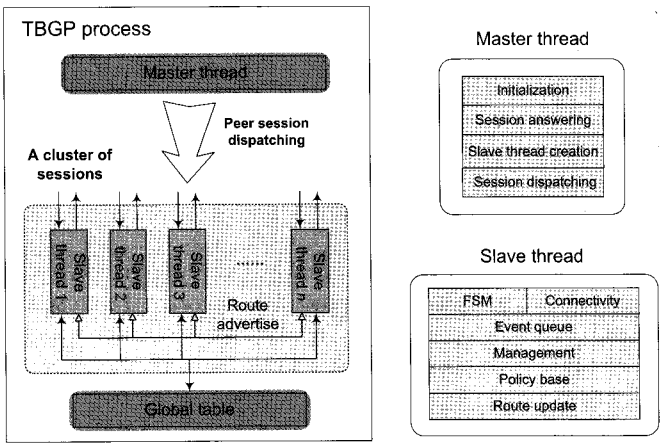


Fig. 1. Architecture of TBGP.

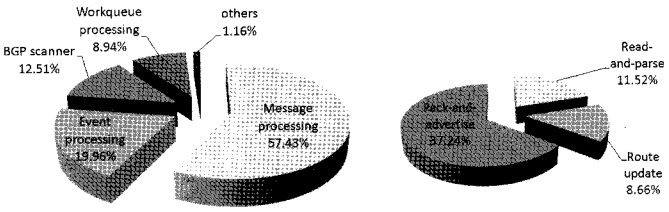


Fig. 2. (a) Percentage of the primary phases in BGP; (b) percentage of the portions in message processing.

creating slave threads, answering session requests and evenly distributing peer sessions among different slave threads. It monitors and answers the connection requests from neighbor sessions, dispatches a slave thread for each new session, and then assigns a socket address to the specified slave thread. In order to balance the session overheads of different threads, the master thread adopts a round-robin dispatch scheme in its implementation. The slave thread is the actual execution entity for a cluster of sessions. Each slave thread is solely responsible for maintaining the finite state machine (FSM) operations, keeping session connectivity, processing update messages, and managing protocol behaviors. It also has its own event queue, so that the multiple sessions triggered by events of different threads may work in parallel. In Fig. 1, each thread receives the routes from its corresponding neighbor sessions and filters them according to the input/output policies from the BGP routing information base (RIB). When routes have been processed by a local slave thread, they also need to be propagated to other slave threads for announcing to all neighbor sessions, maintaining behavior consistent with BGP. In particular, TBGP deploys a shared routing table to keep a consistent route view for all threads, ensuring selection and advertising of the globally optimal routes. With multi-core systems, access to the shared routing table supports high-bandwidth communication and fast synchronization, thereby addressing the synchronization problem of distributed route storage [18] and providing a better aggregated performance.

By utilizing the underlying parallelism in BGP, TBGP introduces a distributed structure and balanced division scheme to create multiple threads with equal numbers of sessions, which

are processed in parallel. This accelerates the message process and provides scalability of the neighbor sessions. Because all the threads in TBGP must have exclusive access to routing table to ensure the consistency of route information, a mass of contentions restricts the protocol efficiency when multiple threads race to access the routing table. To facilitate the quantitative analysis, the performance of TBGP with a shared routing table is abstractly evaluated. It is assumed that the average runtime for each route in BGP consists of three parts: parse time t_p, update time t_u and advertise time t_e. The speedup of TBGP can then be described as:

$$\text{Speedup} = \frac{(t_p + t_u + t_e)n_r + n_s E}{(t_p + t_u^{'} + t_e)\frac{n_r}{n_t} + \frac{n_s E}{n_t} + n_r C} \quad (1)$$

where n_s peer sessions are dispatched to n_t threads to process n_r routes, $t_u^{'}$ denotes the average update time per route in TBGP, E denotes the overhead of maintaining FSM and processing events for a peer session and C denotes the overhead of exchanging a single updated route among multiple threads. Through the Vtune™ [26] toolkit, the BGP profile is first analyzed by capturing the critical phases. The percentages of the primary phases in BGP are summarized in Fig. 2. Message processing accounts for nearly two thirds of the total runtime; most of it is concentrated on the route update phase which is the critical phase of the entire BGP. Generally, introducing exclusive accesses to the routing table will cause most threads to be blocked in the route update phase for a long time, and thus we can predict a larger percentage of $t_u^{'}$ for TBGP than for the original BGP. Assuming that the parameters n_t and n_s invariant, the speedup in (1) inclines to $(t_p+t_u+t_e)n_t/(t_p+t_e+t_u^{'}+n_tC)$ with increasing n_r. Since t_p and t_e keep almost the same value as in BGP, it can be observed that the TBGP performance is mainly determined by the parameter $t_u^{'}$ according to the Amdahl's law, i.e., the efficiency benefits of TBGP could be increased by decreasing $t_u^{'}$ and vice versa. In this context, the significant route-updating bottleneck of TBGP will rebate the performance improvement by multi-threading. To cope with this situation, we propose a highly-efficient parallel table access approach, which reconstructs the routing table and its operations to achieve concurrent table access, thus reducing the parallel access contentions as well as accelerating the route update.

III. FAST PARALLEL TABLE ACCESS FOR TBGP

According to the analysis above, the key point to improve the TBGP performance is to increase the average route update rate. In this section, we first propose a novel table framework with two-level tries to support the fast table access by multiple parallel threads. Then, a heuristic-based divide-and-recombine algorithm is developed to divide the global routing table into several subtrie sets and construct the two-level tries. By evenly distributing the table accesses of multiple threads over different subtrie sets, the access contentions towards the routing table can be reduced.

Let us consider the traditional routing table used in the early stage of TBGP design. To ensure the consistency of route information across multiple threads, we employ a lock mechanism in our design to provide atomic and mutually exclusive table access through two alternative methods. First, we used a global table lock, where all threads share a unified lock and each thread must acquire this lock before accessing the table. In this case, sequential table access when racing for the global lock always resulted in multiple threads being blocked in the route update phase, which appeared to be a major bottleneck as the number of peer sessions or advertised routes scaled up. The other alternative was that each thread only locked the targeted table node and its information. In theory, this could significantly reduce the granularity of lock by allowing multiple threads to lock different nodes in parallel, with the exception of racing for the same nodes, and thereby yielding better performance intuitively. However, this approach led to inconsistency of update behaviors in many cases; for instance, the median node on the traversal path of the current thread would be deleted by other threads, or the current inserted/deleted nodes would be passed by other threads. Therefore, we also considered locking every traversed node for each operation, but this introduced a huge amount of overhead for many lock operations. Taking an insert operation with 20 traversed nodes for example, and supposing that the runtime of insert and lock operations were 2.4 μs and 2.2 μs, respectively, adding one node to the trie would consume no less than 46 μs, which was beyond the average BGP route update time of 33.5 μs. This method was also infeasible in TBGP because of its apparent lock overhead.

To choose a practical method with less overhead, the global table locking was preferred to maintain the correct behavior of parallel table access in the early design. However, with the higher capabilities of multi-cores, the great latency incurred by the sequential route update became more and more serious due to the increasing number of TBGP threads. We reconsidered the routing table and adopted the fine-grain subtries so that multiple threads locking an individual subtrie could access non-overlapping portions of the table to achieve parallel accesses. Then, the effect of access parallelism also relies on the table partition algorithms, which can be further divided into static and dynamic approaches. The static approach decomposes the global table statically at initiation and each subtrie corresponds to this fixed structure. This method is simple and consumes little additional time, but it always suffers from an access imbalance of subtries because it ignores the dynamic characteristics of the node accesses, thus still incurring many contentions. In the dynamic approach, subtries are dynamically partitioned according to the recent access distributions. Through the balance of accesses to different subtries, the contentions could be reduced significantly, as shown in subsection IV-B.2. Here, two different lock entities, permanent and temporary locks, are taken into account in the implementation of the dynamic scheme. The permanent lock is created at the beginning of the program and is only destroyed when the program terminates, while the temporary lock is dynamically created and destroyed after each table partition. The temporary lock always significant increases the overhead of the partition algorithm because it incurs more lock create and destroy operations while partitioning subtries. Conversely, the permanent lock is always preferred because it saves locking cost through the assignment of locks to different subtrie sets. In summary, locking subtries with the dynamic partition eliminates the sequential route update caused by locking

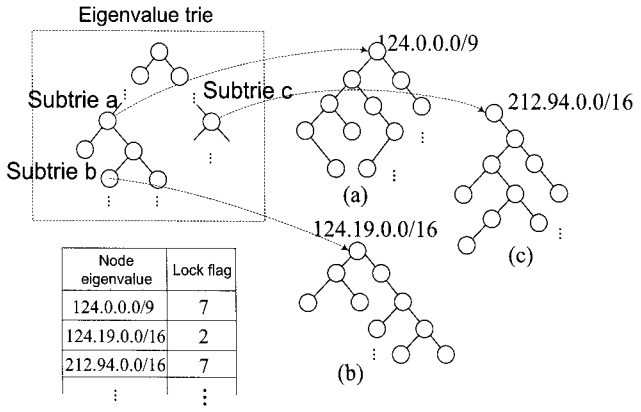


Fig. 3. Architecture of two-level tries table.

the global table and the great overhead caused by locking single table node. It benefits from fewer contentions than the static partition by distributing parallel accesses dynamically, and it also favors a permanent lock entity to achieve faster partition operations.

A. Two-Level Tries Table Framework

The two-level tries table adopts a dynamic partition algorithm to decompose the traditional routing table into multiple subtries, as shown in Fig. 3. The first level trie, named the eigenvalue trie, is organized with the roots of partitioned subtries so that the matched subtrie can be located quickly and locked individually without any influence from accessing of other subtries by different threads. In the first level trie, each eigenvalue node corresponds to a unique subtrie in the original routing table and records its IP prefix and prefix length as eigenvalues. Then, the second level trie is a set comprising all the partitioned subtries. By decomposing the global table, the two-level tries table empowers threads to access different subtries in parallel and favors the routines (e.g., TBGP scanner) to access throughout the routing table. A typical two-level tries access process is described as follows: The thread searches in the eigenvalue trie to longest perfix match a particular eigenvalue node, then locks the corresponding subtrie, if not previously locked by another thread, and accesses the subtrie exclusively. Since the eigenvalue trie is shared to be read by all threads without any interference, the two-level tries table fulfills the demand for parallel table access.

The performance improvement of parallel table access with the two-level tries table is primarily due to the partition algorithm balancing subtrie accesses to minimize parallel access contentions. To the best of our knowledge, the common trie partition algorithms always suffer from subtrie access imbalance, causing many access contentions. Meanwhile, the dynamic partition algorithm is designed to complete in a short period, allowing fast route processing. Thus, we put forward a novel fine-grain subtrie partition algorithm, after which several subtrie sets are recombined with the goal of evenly distributing accesses over different sets. Each set includes multiple subtries of various sizes and is granted a private lock that is shared by all the subtries in the set. With the private lock, locking an arbitrary subtrie will also lock other subtries in the same set. For instance, in Fig. 3, subtrie a and c belong to the same set and share the private lock whose identifier is 7. The 7th lock is first acquired when accessing subtrie a, and subtrie c is also locked at the same time.

B. Heuristic-Based Divide-and-Recombine Algorithm

In this section, we introduce a dynamic table construction algorithm to periodically update the two-level tries table according to the changes in nodes and their access frequencies. The purpose of reconstructing the routing table by dynamic partitioning is to relieve the parallel access contentions in TBGP, thereby achieving a fast route update of multiple threads. The dynamic partition algorithm emphasizes high efficiency. In the projected prototype, the execution of our proposed algorithm is triggered at particular intervals, and it will lock the entire eigenvalue trie to break off all thread accesses to the table. In order to ensure fast route updates, the table construction must be completed in a short period.

First, let us specify the optimization goals of our algorithm. The heuristic-based divide-and-recombine algorithm partitions the traditional routing table into a large number of fine-grain subtries and then recombines them into a few sets according to the subtrie access frequencies, i.e., the access frequencies. Intuitively, if the partitioned subtries are distributed evenly into different sets, the probability that threads simultaneously race to the same subtrie will be decreased, therefore reducing the access contentions towards the routing table. To facilitate the above analysis, we first partition the traditional routing table into n subtries and assume the access level of the ith $(1 \leq i \leq n)$ subtrie to be $f(i)$ which sums up all the access frequencies of the subtrie nodes. Then, to balance the thread accesses to different sets, we allocate n partitioned subtries into M individual sets and attempt to reduce the gap between the maximal and minimal access levels of the sets, as shown in (2), where the access level of set $Q_j (1 \leq j \leq M)$ sums up all the access levels of the subtrie elements. Note that the optimal number of subtrie sets M is not chosen at random and always needs to be evaluated under different numbers of slave threads n and routes R. A larger M decreases the probability of colliding at the same set but also makes the divide-and-recombine algorithm become time-consuming. The evaluation details are given in section IV.B.2.

$$\text{minimize}(\max F(Q_j) - \min F(Q_j)) \tag{2}$$

where $F(Q_j) = \sum_{i \in Q_j} f(i), 1 \leq j \leq M$.

The divide-and-recombine algorithm is composed of two phases. The task of the first phase is to logically divide the routing table into fine-grain subtries and yield the eigenvalue trie structure for indexing them. The detailed division algorithm is depicted in algorithm 1, which takes the routing table as an input. According to the access information of the routing table (e.g., the access frequency of each node and the access levels of children branches), the algorithm makes a recursive trie traversal, constantly dividing the routing table into subtries and producing the eigenvalue trie nodes until the root node is visited. For simplicity, we begin by considering a certain node as the current one. Lines 5–6 of algorithm 1 first visit the current node's left and right branches by calling the function *post_order_leaf*. Through traversals in the post-order, the

Algorithm 1: Fine-grain trie division.

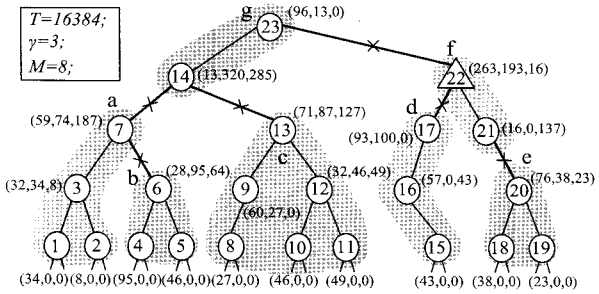
```
01 int post_order_leaf (trie node) {
02 if (node = NULL) return (0);
03 else {
04   if ((node.lp ≠ NULL) || (node.rp ≠ NULL)) {
05   node.lef ← post_order_leaf(node.lp); // Left branch
06   node.rig ← post_order_leaf(node.rp); // Right branch
07   record ← 0; aver ← T/(2^γ M);
08   if((node.local≤aver)&(node.local+ node.lef
                                +node.rig > aver)){
09     if ((node.lef ≥ (1/2)aver) & (!Signed(node.lp))
10     {ToSign (node.lp); record ← record-node.lef;}
11     if ((node.rig ≥ (1/2)aver)&(!Signed(node.rp))
12     {ToSign (node.rp); record←record-node.rig;}
13     if (node==ROOT) {ToSign (node); return(0);}
14     return(record+node.lef+node.rig+node.local);}
15   if (node.local > aver) { // If it is a hot node
16     if ((node.lef≥(1/2)aver)&(!Signed(node.lp))
17     ToSign(node.lp);
18     if ((node.rig≥(1/2)aver)&(!Signed(node.rp))
19     ToSign (node.rp);
20     ToSign(node);
21     return(0);}
22   if (node==ROOT) { ToSign(node); return(0); }
23   return(node.lef+node.rig+node.local);
24 }}
```

partitioned subtries in its children branches are cut from the current node and their roots are signed as eigenvalue nodes and inserted into the eigenvalue trie. Next, we investigate how to divide these nodes into a number of subtries associated with the current node. Let M be the number of subtrie sets; the average access level of produced subtries should be close to be $\frac{T}{2^\gamma M}$, where 2^γ describes the average size of the subtrie set and T indicates the total access level of the table, determined by summing all the node access frequencies. In algorithm 1, the access frequencies of most normal nodes are less than $\frac{T}{2^\gamma M}$ with the exception of a few hot nodes that have higher access frequencies. The atomic property of these hot nodes is bad for producing the fine-grain subtries, thus hampering the access balance among different sets. In this algorithm, we name the subtrie with the root of a hot node as a hot subtrie, distinguishing it from the normal ones, and we tend to decrease the hot subtrie size to reduce this adverse effect. In lines 8–14, we first examine whether the current node is a normal node and the access level of the subtrie with the root of the current node is more than $\frac{T}{2^\gamma M}$. If both conditions are satisfied, we further look at its children branches. In lines 9–12, the children branches with access levels more than $\frac{T}{2^{\gamma+1}M}$ are separated to be new subtries by calling the function $ToSign$. If the access levels of the children branches are both less than $\frac{T}{2^{\gamma+1}M}$, no new subtries will be generated at this recursive level. The trie with the root of the current node will be separated in the recursion function of its father node, since its access level is between $\frac{T}{2^\gamma M}$ and $\frac{T}{2^{\gamma-1}M}$. Next, lines 15–21 take into account the situation in which the current node is a hot one. They consider the hot node as well as any children branches with access levels more than $\frac{T}{2^{\gamma+1}M}$ to be new subtries, and produce their eigenvalue nodes. This algorithm is executed recursively to produce subtries and eigenvalue trie nodes. With algorithm 1, the access levels of normal and hot subtries follow the range $[\frac{T}{2^{\gamma+1}M}, \frac{T}{2^{\gamma-1}M}]$ and $[S, S+\frac{T}{2^\gamma M}]$ $(S > \frac{T}{2^\gamma M})$, respectively, and the upper limit for the number of partitioned subtries L is $\frac{T}{2^{\gamma+1}M}$ as shown in (3).

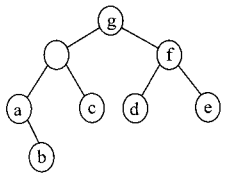
$$L \le \max\left\{\frac{T}{T/(2^{\gamma+1}M)}, \frac{T}{S}\right\} < 2^{\gamma+1}M \tag{3}$$

Fig. 4 gives an example of the division process for a routing table including twenty-three prefix nodes. Let the parameters T, M, and γ be 16384, 8 and 3, respectively. The division algorithm is used for dividing the compressed binary trie into fine-grain subtries. As illustrated in Fig. 4, the circled node denoting the individual prefix is numbered in the postorder. For any given circled node, we use a unique triple $< local, lef, rig >$ to describe its access information, where element $local$ indicates its access frequency, and elements lef and rig, respectively, denote the access levels of its left and right branches. The postorder traversal of our recursive algorithm begins with node 1. By inspecting the node access frequency and summing all the elements of its triple, neither of the conditions at lines 8 and 15 can be met, so line 23 directly returns the sum of its triple elements to be the element lef of its father. Next, the algorithm visits the subsequent nodes in postorder and finds that similar situations happen until node 7 is visited. Node 7 is a normal node and the sum of its triple elements is above 256, satisfying the conditions of line 8. To divide the routing table, lines 9–14 of the algorithm further investigate its children branches and separate the right branch, with an access level of 187, to be a new subtrie. After that, the algorithm traverses nodes from 8, and the conditions at lines 8 and 15 remain false until node 13 is visited. In node 13, the sum of the triple elements equals 285 but neither its children branches have access level of 128; therefore it does not allow pruning of the branches. The large trie with the root of node 13 should be paid more attention. When traversing in postorder to node 14, the right branch with the root of node 13 satisfies the condition of line 11, so it is cut to a new subtrie. Similar situations happen at the left branch of node 14 and the right branch of node 21, where the branch with an access level above 128 is pruned to be a new subtrie. However, when the algorithm traverses to node 22, its access frequency of 263 satisfies the condition at line 15 and proves that it is a hot node. In this case, lines 16–17 mark its left branch with an access level of 193 to be a new subtrie. Then, in line 20, the trie with the root of node 22 is also signed to be a new subtrie. The recursive algorithm works until the current node is equal to a global variable, ROOT, which denotes the root pointer of the trie. The algorithm terminates after visiting node 23 because the condition at line 22 is met. The partitioned subtries as well as the created eigenvalue trie are shown as Figs. 4(a) and 4(b), respectively.

The second phase is to allocate subtries from the first phase into M sets so as to evenly distribute the accesses over different subtrie sets and relieve contentions. To exploit the parallelism of the routing update, the subtries from different sets are



(a)



a	10.32.0.0/11
b	10.62.0.0/15
c	10.120.0.0/13
d	10.208.0.0/12
e	10.248.0.0/13
f	10.192.0.0/10
g	10.0.0.0/8

(b)

Fig. 4. Example of the fine-grain trie division: (a) Division process for the routing table and (b) eigenvalue trie.

Algorithm 2: subtrie recombination algorithm.

01 Sort the access frequencies of subtries as
$f(1) \geq \cdots \geq f(n)$, where, $n \leq 2^{\gamma+1}M$.
02 $F(Q_j) \leftarrow 0 (j = 1, \cdots, M), i \leftarrow 1$;
03 $r_1[0] = \cdots = r_M[0] = 0; p_1 = \cdots = p_M = 0$;
/* r_j denotes the subtries array of Q_j;
p_j denotes the top pointer of r_j.*/
04 $\alpha \leftarrow$ Min_$IDX(F(Q_1), \cdots, F(Q_M)), 1 \leq \alpha \leq M$
05 $\beta \leftarrow$ Max_$IDX(F(Q_1), \cdots, F(Q_M)), 1 \leq \beta \leq M$
06 **if** $((F(Q_\beta) - f(r_\beta[p_\beta])) > F(Q_\alpha))$ {
/* after removing the top subtrie $r_\beta[p_\beta]$, the
access level of Q_β is bigger than $F(Q_\alpha)$ */
07 $Q_\beta \leftarrow Q_\beta - \{r_\beta[p_\beta]\} + \{i\}; Q_\alpha \leftarrow Q_\alpha + \{r_\beta[p_\beta]\}$;
/* exchange operation, put subtrie i and $r_\beta[p_\beta]$
to set Q_β and Q_α, respectively*/
08 $p_\alpha \leftarrow p_\alpha + 1; r_\alpha[p_\alpha] \leftarrow r_\beta[p_\beta]; r_\beta[p_\beta] \leftarrow i$;
/* modify the top of subtries array r_α, r_β*/
09 **if** $(f(r_\alpha[p_\alpha - 1]) < f(r_\alpha[p_\alpha]))$
10 $\{tmp \leftarrow r_\alpha[p_\alpha]; r_\alpha[p_\alpha] \leftarrow r_\alpha[p_\alpha - 1]; r_\alpha[p_\alpha - 1] \leftarrow tmp;\}\}$
11 **else** $\{Q_\alpha \leftarrow Q_\alpha + \{i\}; p_\alpha \leftarrow p_\alpha + 1;\}$
/*otherwise, current subtrie i is put into Q_α*/
12 **if** $(i \leq n)$ $\{i \leftarrow i + 1$, goto line 03$\}$ **else finish**.

allowed to be accessed simultaneously by multiple threads. The recombination algorithm is illustrated in algorithm 2, where the subtrie set j is represented by Q_j and the access levels of subtrie i and subtrie set j are indicated by $f(i)$ $(1 \leq i \leq L)$ and $F(Q_j)$ $(1 \leq j \leq M)$. Before the recombination phase we first sort the subtries from the first phase in descending order of $f(i)$, and thus we refer to the subtries by their assigned number. Then, after initializing some temporary variables, such as r_j and p_j, the functions Min_IDX and Max_IDX at lines 4–5 are used to return the lower indices of the minimum and maximum access level $F(Q_j)$ as α and β, respectively. The basic idea of algorithm 2 can be summarized as putting the current subtrie into the set Q_α at line 11, thereby limiting the increment of $F(Q_\beta) - F(Q_\alpha)$ and balancing the access level in each subtrie set. However, this method belongs to a greedy algorithm, and the produced result may still have room for further optimization. In this section, we give an additional simple but efficient method to further balance the accesses among different sets. The further balance can be achieved by exchanging the specified subtries of Q_α and Q_β which respectively correspond to the subtrie sets with minimal and maximal access level. More specifically, if the condition $F(Q_\beta) - f(r_\beta[p_\beta]) > F(Q_\alpha)$ is satisfied before dispatching the subtries, the subtries $r_\alpha[p_\alpha]$ and $r_\beta[p_\beta]$ should be exchanged to further reduce the difference between $\max\{F(Q_j)\}$ and $\min\{F(Q_j)\}$. Let Q_j and Q_j^* denote the access levels of set j before and after the exchange, respectively; the increased balance can be confirmed as follows.

$$
\begin{aligned}
&|F(Q_\beta^*) - F(Q_\alpha^*)| \\
&= F(Q_\beta) - f(r_\beta[p_\beta]) + f(i) - (F(Q_\alpha) + f(r_\beta[p_\beta])) \\
&= |(F(Q_\beta) - f(r_\beta[p_\beta])) - F(Q_\alpha) + f(i) - f(r_\beta[p_\beta])| \\
&< |(F(Q_\beta) - f(r_\beta[p_\beta])) - F(Q_\alpha)\| + \|f(i) - f(r_\beta[p_\beta])| \\
&= (F(Q_\beta) - f(r_\beta[p_\beta])) - F(Q_\alpha) + f(r_\beta[p_\beta]) - f(i) \\
&= F(Q_\beta) - (F(Q_\alpha) + f(i)) \\
&= |F(Q_\beta) - (F(Q_\alpha) + f(i))|
\end{aligned}
$$

It is obvious that the access level difference between the two sets is reduced since $\|F(Q_\beta^*) - F(Q_\alpha^*)\| < \|F(Q_\beta) - (F(Q_\alpha) + f(i))\|$, and we consider $\|F(Q_\beta) - (F(Q_\alpha) + f(i))\| - \|F(Q_\beta^*) - F(Q_\alpha^*)\|$ to be the optimized value of the exchanging operation that suits the optimization objective given in (2).

Since the division process needs to travel throughout all the table nodes and generate the eigenvalue nodes to organize the first level trie, the complexity of the first phase is $O(N)$. On the other hand, because the upper limit for the number of subtries from the first phase is $2^{\gamma+1}M$ and each iteration will execute a compare operation, the recombination process requires $O(M^2)$ time. Therefore, the total time complexity of our divide-and-recombine algorithm is $O(N + M^2)$.

C. Modifications for Table Operations

In contrast to the traditional routing table, our scheme employs a method of two-level routing table access. The table operations of lookup, insert, delete, and route update, beginning with the process of longest prefix matching (LPM), are further divided into two stages to suit our two-level tries. The eigenvalue LPM is used to index the target eigenvalue trie node first. Once a certain eigenvalue node is found, the LPM node can be further located inside the indexed subtrie. However, some incoming prefixes may exist that do not match any eigenvalue node when applying the searching process, and therefore result in a table operation exception. In this context, we refer to the objects of these operations as dissociative nodes. The dissociative nodes always act as ancestor or sibling nodes of the top subtrie indexed by the eigenvalue trie root. In order to operate consistently with the behaviors of traditional table, these dissociative nodes are ar-

Table 1. Complexity comparison with typical algorithms.

Algorithm	Lookup	Insertdelete	Memory	Trie update
Binary trie	$O(W)$	$O(W)$	$O(NW)$	NA
Path-compressed trie [30], [31]	$O(W)$	$O(W)$	$O(N)$	NA
k-stride multibit trie [32], [33]	$O(W/k)$	$O(W/k+2^k)$	$O(2^k NW/k)$	NA
Prefix-length based binary trie [34]	$O(\log_2 W)$	$O(N\log_2 W)$	$O(N\log_2 W)$	NA
Two-level tries	$O(W/v+v)$	$O(W/v+v)$	$O(N+M)$	$O(N+M^2)$

ranged into a special dissociative subtrie whose root is actually the root of the second level trie. For any incoming prefix, if a particular eigenvalue node is found by the longest prefix matching process, second level searching in the target subtrie will be performed; otherwise, the LPM will begin from the root of the dissociative subtrie indexed by the second level trie root, thus ensuring the correctness of operations. Using this basic principle, we then properly modify the table operations to make them simple and effective.

IV. PERFORMANCE EVALUATION

A. Complexity Analysis

The performance of the proposed routing table, in terms of the operation runtime and memory overhead, is first theoretically compared with the best known algorithms as shown in Table 1. The two-level tries table employs two-level access to the eigenvalue trie and subtries, and the runtimes of the lookup, insert, and delete operations have a complexity of $O(W/v+v)$, where W denotes the prefix length and v is the average depth of the subtries. Correspondingly, the time complexity of the lookup operation for a prefix-length-based binary search scheme is $O(\log_2 W)$, but the insert and delete operations are more complicated, with a time complexity of $O(N\log_2 W)$. The k-stride multi-bit trie can look up a node within a period of $O(W/k)$, but its redundant nodes always lead to the high time complexity of $O(W/k+2^k)$ for insert and delete operations. Our approach, which incurs the extra memory of the eigenvalue trie and recorded access frequencies, only has a memory complexity of $O(N+M)$. This is much less than those of the binary trie, k-stride multibit trie, and prefix-length based binary trie, but slightly more than that of the path-compressed trie, which has the optimal memory consumption of $O(N)$. Here, the number of subtrie sets M is far less than the number of table nodes N in our proposed approach, and thus the memory increment in comparison to the path-compressed trie scheme is negligible. In addition, the dynamic table construct operation introduced by our approach needs to repartition global table and reorganize the eigenvalue trie, which has a time complexity of $O(N+M^2)$. To achieve faster operation, this construction process is implemented in the BGP scanner module and is always activated with a long time interval, thereby saving the time of traversing the table and reducing the extra cost for the route processing.

B. Simulation Results

Let us consider the TBGP based on Quagga 0.99.9 [27] that provides a high quality, multi-server routing engine involving various routing protocols. In the Quagga software package, its BGP daemon is responsible for handling the projected TBGP protocol and interacting with the zebra daemon that is used for routing table management and route redistribution. In this section, we modify the routing table structure and table operations of TBGP to support the parallel table accesses to two-level tries. By adjusting the number of threads, peer sessions and advertised routes, we use the AX4000 series [28] to verify the correctness of the proposed parallel access approach by comparing the updated results of the two-level tries table with the traditional routing table. Then, we run the TBGP protocol with a prefix database of routing table snapshots extracted from RouteViews [29], and we compare the two-level tries with the traditional routing table in terms of parallel access contentions, update time per route, maximal update time of thread, route convergence time, and update message throughput. All the experiments are performed on a dual quad-core Xeon server with Linux 2.6.18-8AX operating system.

B.1 Parallel Access Contentions

The experimental work begins by analyzing the parallel access contentions in the traditional table and two-level tries table to validate the contention reductions of our approach. We extract route entry snapshots, such as the BGP routing data on Oct.1 2008 from RouteViews, to measure the performance improvement. The data are collected in two forms: RIB files and update files. The RIB files contain the contents of all the BGP neighbors and are used for routing table initialization. The update files contain new updates from BGP neighbors and are used for route updates. We select the number of routes, ranging from one million to ten million, and establish a number of peer sessions through AX4000. According to the previous hops of the AS path, all the routes are distributed over different sessions so that any session only accommodates the routes with the same previous hop. Then, we specifically consider the number of access contentions. For the traditional table, a global lock of the entire routing table inevitably leads to exclusive access by different threads, and simultaneous accesses by arbitrary two threads will cause a collision. But for the two-level tries table, a private lock is granted to each subtrie set and the contentions occur only when different threads collide at the same set. To further collect the contention number of each subtrie set, a separate counter is assigned for each private lock in our experiments. The counter of a particular subtrie set is initialized to zero and will be increased by one if a certain thread succeeds in locking it. Subsequently, this counter is increased by one when any other thread attempts to access any subtrie of this set, before the cur-

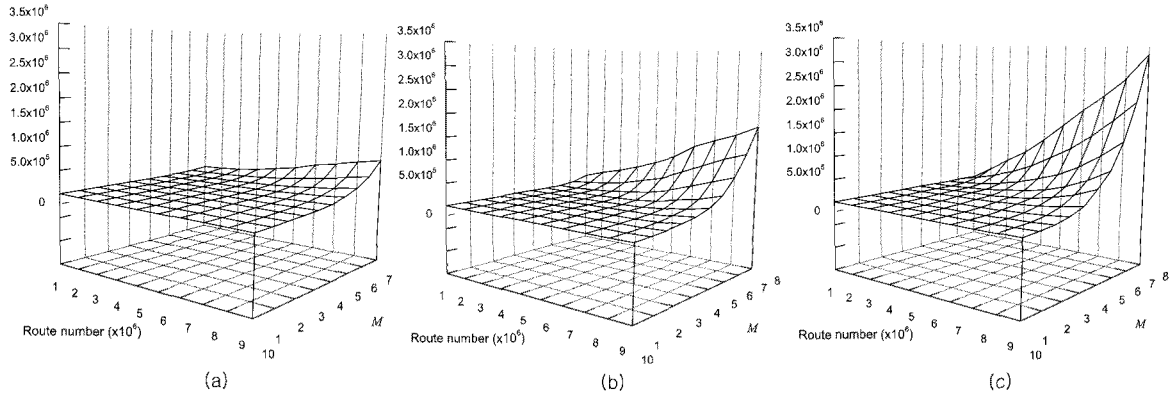


Fig. 5. Contentions under different M and R when (a) n is 2; (b) n is 4; and (c) n is 8.

rent thread finishes its table operation. When the lock is released by the current thread, its counter will be cleared and added to the total number for that set. If k threads colliding at the same subtrie set compete for access, only one candidate can be granted access at a time, and thus the number $k-1$ will be added to the total contention number of the subtrie set.

In our experiment, the divide-and-recombine algorithm is implemented in the TBGP scanner module, where the parameter T is collected by summing up all the node access frequencies, and the average access level of subtries is designated to be $T/256M$ since an average subtrie number of 256 per set is enough to balance the thread accesses to different sets based on past experience. Once the TBGP with our proposed algorithm and table operations is designed, we use it to perform the experiment by varying the number of slave threads n. The parallel access contentions with the traditional table and two-level tries table with n values of 2, 4, and 8 under different subtrie sets M and route numbers R are illustrated in Fig. 5. It can be observed that the contentions with the traditional table and M equal to one increase sharply with increasing n and R. When n is 2, the number of contentions grows slightly (Fig. 5(a)), but in sharp contrast, when n is increased to 8, the number of contentions grows sharply with increasing route number (Fig. 5(c)), even reaching 334% of the total routes in the worst case. Increasing the thread number has apparently exposed the performance bottleneck of TBGP when accessing the traditional table. However, if we use the two-level tries table, a significant decrease in contentions can be observed in all cases. In particular, as shown in Fig. 5, the probability of colliding at the same set continues to decrease even with increasing M. Further increasing the number of subtrie sets can effectively improve the TBGP performance by reducing the number of parallel access contentions and the average subtrie access probabilities, but it also incurs other issues, e.g., making the divide-and-recombine algorithm become time-consuming. In practice, the execution of our divide-and-recombine algorithm needs to halt all the table accesses of slave threads, and thus the overlarge M will increase the runtime of the algorithm. This results in a large number of update packets not being processed in time or even being lost, leading to low processing efficiency. In our projected future prototype, selecting a modest value of M with the premise of decreasing the large number of parallel access contentions is necessary. According to the evaluated results in Figs. 5(a), (b), and (c), the decrease in contentions tends to be saturated when M reaches 8, 12, and 16, respectively, with significant reductions of 87.3%, 92.8%, and 97.3% in comparison with those in traditional table. Therefore, we consider that the optimal number M of 8, 12, and 16 will obtain the best effects when the thread number is configured as 2, 4, and 8, respectively.

B.2 Performance for Route Update

In this section, the performance of route updates in the cases of the traditional table and two-level tries table are evaluated in terms of update time per route (UTPR) and maximal update time of thread (MUTT). The former metric indicates the route update runtime involving the average locking operation cost of 2.2 μs, and the latter one represents the maximal runtime of route update phases in different threads. Following the experiments in subsection IV-B.1, the UTPRs continue are sampled when 2, 4, and 8 threads configured with 8, 12, and 16 subtrie sets, respectively, process ten million routes. The statistical results are summarized in Fig. 6. With the traditional table and M set to be one, the UTPRs of two slave threads concentrate in the ranges of 30–40 and 40–50μs. However, in the case of 4 or 8 threads, the distribution of UTPRs is dispersed and many samples are moved to the ranges of 40-50 and 50-60μs, and 3% even go beyond 80μs when n equals 8. The primary reason is that the parallel access contentions always make threads blocked during the route update phase by sequentially accessing the routing table; consequentially, the blocking time and route update runtime become larger and larger with increasing thread candidates. On the contrary, the performance improvement of the two-level tries table scales well with the number of threads, as shown in Fig. 6, where most UTPRs are concentrated in the ranges of 20–30 and 30–40 μs in all cases. With an increasing number of threads, we can increase the subtrie set number M to support parallel accesses to the different sets, and thus it can be predicted that the route update of each thread would be seldom blocked even when a large number of threads compete for the routing table.

The MUTTs with configurations of 2, 4 and 8 threads are collected with Vtune^{TM} toolkit by regulating route numbers from one million to ten million. As shown in Fig. 7, the MUTTs of

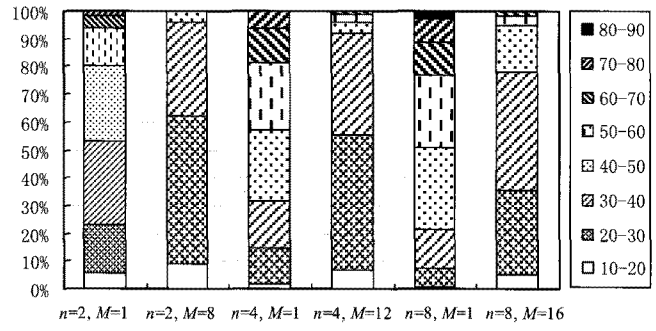


Fig. 6. Distribution of sampled UTPRs.

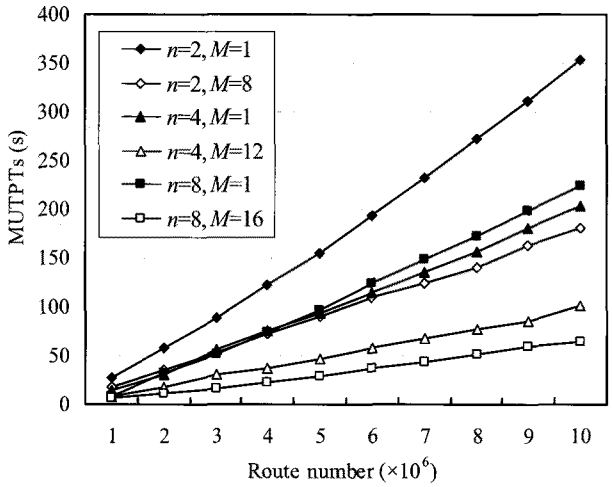


Fig. 7. Statistical MUTTs of traditional table and two-level tries table.

the traditional table increase sharply with linear growth of the number of routes, making it difficult to support scalability of the thread number. For example, the MUTT of 8 threads exceeds that of 4 threads even if the former processes fewer routes, revealing that the TBGP route update performance with the traditional table declines with increasing n. On the other hand, when configured with the two-level tries table, the MUTTs increase slightly with increasing route number, but they are reduced by 48.7%, 50.5%, and 71.1% compared to the traditional table. The two-level tries table delivers good improvement on route updates with increasing number of threads, therefore addressing the scalability problem of thread number; e.g., the MUTT of 8 threads is less than that of 4 threads by 63% on average.

To reduce the overhead of traversing the routing table, our proposed algorithms are triggered periodically in the TBGP scanner module, but they inevitably bring some overhead for dynamically constructing the two-level tries table. When processing ten millions routes under 2, 4, and 8 threads, the total runtime captured by Vtune™ toolkit for periodically regulating the two-level tries consumes averages of 0.52, 0.84, and 1.4 seconds, which are far less than the decreases in the MUTTs. Therefore, it can be said that the heuristic algorithm has almost no adverse effect on the TBGP execution. In addition, since the number of nodes in the eigenvalue trie of algorithm 1 is strictly constrained by $2^{\gamma+1}M$, the memory consumption by the first level trie with 16 sets is measured to be 320 KB, nearly 0.03%

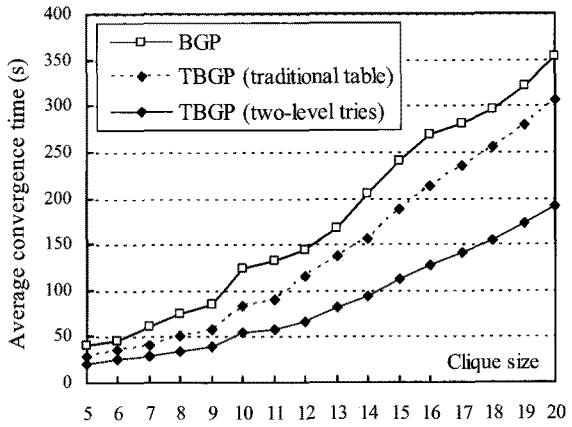


Fig. 8. The comparisons of convergence time for BGP and TBGPs.

of the traditional table size.

B.3 Route Convergence

The full-link topology of ASs with clique sizes from 5 to 20 is simulated by SSFNet [35], which comprises a collection of components for modeling and simulation of Internet protocols and networks at or above the IP packet level. In this section, the BGP4 component of SSFNet is replaced by TBGP with four slave threads to evaluate the route convergence of TBGP. Each AS only includes one router, the transmission delay of the links among routers is set to be constant at 0.01s, and the default minimum route advertisement interval (MRAI) of 30s is used. To avoid the centralization of route messages received the by routers, we then add stochastic oscillations in the detailed simulation. In the initial state, each router has a stable route to the specified destination network, and the routes are withdrawn by source nodes to invoke route convergence. We collect the average convergence times of the original BGP, TBGP with the traditional table, and TBGP with two-level tries, as shown in Fig. 8, by performing the simulation on each clique size and creating different generated random seeds in each simulation. When scaling up with clique size, the convergence time of the original BGP incurs a significant increase, while those of TBGP are improved. This improvement can be attributed to the faster route processing by multithreads. For the original BGP, each router updates the routes in order, so the convergence time goes up rapidly as the updated route or clique size increases. For TBGP with the traditional table, the convergence time with clique sizes from 5 to 10 has a median decrease because the synchronization of the routing update procedure is accelerated by multiple route processing threads. However, this acceleration decreases when the clique size exceeds 10, since more routing table accesses block the threads in the route update phase. The convergence time reduction of TBGP with the traditional table versus that of traditional BGP gradually decreases with clique sizes above 10, e.g., it drops from 33.2% to 13.2% when increasing the clique size from 10 to 20. The primary advantage of TBGP with two-level tries is that it solves the mass of contentions. Since the multithreaded route update and routing update synchronization processes are significantly accelerated, the convergence time of TBGP with two-level tries presents a

noticeable decrease in Fig. 8, where the reduction in the convergence time achieves nearly 49.7% versus that of the original BGP, on average.

V. CONCLUSIONS

Exploiting the thread-level parallelism in BGP helps improve its performance to satisfy the increasing number of applications of the Internet, but the TBGP performance improvement is still restricted by a mass of contentions when racing to access the shared routing table. For the projected future TBGP, our key insight is to relieve these contentions, thereby supporting the fast parallel route update to achieve ultra-high route processing speed. In this paper, we present a novel table structure of two-level tries and devise a heuristic-based divide-and-recombine algorithm to yield the table structure dynamically, therefore accelerating the parallel route updates of multiple threads. We then modify the typical table operations and validate their correctness to be consistent with the behaviors of the traditional routing table. The experimental results show that the contentions by parallel access to the routing table decrease sharply by 92.5% on average, the update time per thread is reduced by 56.8%, and the convergence time of BGP update messages is enhanced by about 49.7%.

REFERENCES

[1] Y. Rekhter, T. Li, and S. Hares, "A border gateway protocol 4 (BGP-4)," RFC 4271, 2006.

[2] M. Beesley, "Router/switch control plane software challenges," in *Proc. ACM/IEEE ANCS*, 2006.

[3] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica "Achieving convergence-free routing using failure-carrying packets," in *Proc. ACM SIGCOMM*, 2007, pp. 176–187.

[4] J. Mudigonda, H. M. Vin, and S. W. Keckler, "Reconciling performance and programmability in networking systems," in *Proc. ACM SIGCOMM*, 2007, pp. 15–26.

[5] G. Huston, "The BGP Report for 2005," ISP Column, 2006. [Online]. Available: http://ispcolumn.isoc.org/2006-06/bgp-upds.html

[6] S. Agarwal, C.-N. Chuah, S. Bhattacharyya, and C. Diot, "Impact of BGP dynamics on router CPU utilization," *LNCS* 3015, Springer-Verlag, pp. 278–288, 2004.

[7] A. Feldmann, O. Maennel, and Z. M. Mao, "Locating internet routing instabilities," in *Proc. ACM SIGCOMM*, 2004, pp. 205–218.

[8] A. Feldmann, H. W. Kong, O. Maennel, and A. Tudor., "Measuring BGP pass-through times," *Passive Active Meas. Workshop*, vol. 3015, pp. 267–277, 2004.

[9] C. Labovitz, G. R. Malan, and F. Jahanian, "Internet routing instability," *IEEE/ACM Trans. Netw.*, vol. 6, no. 5, pp. 515–528, 1998.

[10] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and inter-domain routing," *IEEE/ACM Trans. Netw.*, vol. 10, no. 1, pp. 232–243, 2002.

[11] K. Varadhan, R. Govindan, and D. Estrin, "Persistent route oscillations in inter-domain routing," *Compter Netw.*, vol. 32, no. 1, pp. 1–16, 2000.

[12] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed Internet routing convergence," in *Proc. ACM SIGCOMM*, vol. 9, pp. 293–306, 2000.

[13] W. Sun, Z. M. Mao, and K. G. Shin, "Differentiated BGP update processing for improved routing convergence," in *Proc. ICNP*, 2006, pp. 280–289.

[14] J. Karlin, S. Forrest, and J. Rexford, "Pretty good BGP: Improving BGP by cautiously adopting routes," in *Proc. ICNP*, 2006, pp. 290–299.

[15] M. Bjorkman and P. Gunningberg, "Performance modeling of multiprocessor implementations of protocols," *IEEE/ACM Trans. Netw.*, vol. 6, no. 3, pp. 262–273, 1998.

[16] X. Xiao and L. M. Ni, "Parallel routing table computation for scalable IP routers," in *Proc. CANPC*, 1998, pp.43–55.

[17] T. Klockar, M. Hidell, L. Carr-Motyčkova, and P. Sjödin "Modularized BGP for decentralization in a distributed router," Winternet Grand Finale workshop, pp. 72–84, 2005.

[18] X. Zhang, P. Zhu, X. Lu, "Fully-distributed and highly-parallelized implementation model of BGP4 based on clustered routers," in *Proc. ICN, LNCS 3421, Berlin: Springer-Verlag*, 2005, pp. 433–441.

[19] K. Wu, J. Wu, K. Xu, "A tree-based distributed model for BGP route processing," in *Proc. HPCC, LNCS 4208, Berlin: Springer-Verlag*, 2006, pp. 119–128.

[20] K. Xu, H. He, "BGP parallel computing model based on the iteration tree," *J. China Universities of Posts and Telecommun.*, vol. 15, pp. 1–8, Sept., 2008.

[21] L. Gao, Z. Gong, et.al, "A TLP approach for BGP based on local speculation," *Science in China Series F: Information Sciences*, vol. 51, no. 11, pp. 1772–1784, 2008.

[22] X. Huang, S. Ganapathy, and T. Wolf, "A scalable distributed routing protocol for networks with data-path services," in *Proc. ICNP*, 2008, pp. 318–327.

[23] Z. Liang, K. Xu, J. WU, and Y. Chai, "Parallel routing lookup system based on non-overlapping prefix sets," *Acta Electronica Sinica*, pp. 1277–1281, 2004.

[24] T. Srinivasan, M. Sandhya, and N. Srikrishna, "An efficient parallel IP lookup technique using CREW based multiprocessor organization," in *Proc. CNSR*, 2006, pp. 221–226.

[25] K. Zheng, C. Hu, H. Lu, and B. Liu, "A TCAM-based distributed parallel IP lookup scheme and performance analysis," *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 863–875, 2006.

[26] $Vtune^{TM}$, Intel vtune performance analyzer for Linux. Santa Clara (CA): Intel Corporation, 2007.

[27] Qugga project. [Online]. Available: http://ww.quagga.net /download/

[28] Spirent Adtech AX4000 broadband test system. [Online]. Available: http://www.smartechconsulting.com/Adtech-Spirent-AX4000-AX-4000-16-Slot-Rackmount-Chassis

[29] RouteViews project. [Online]. Available: http://archive.routeviews.org/

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

[31] K. Sklower, "A tree-based packet routing table for berkeley unix," in *Proc. Winter Usenix Conf.*, 1991, pp. 93–99.

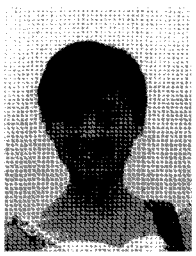
[32] S. Sahni and K. S. Kim, "Efficient construction of multibit tries for ip lookup," *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 650–662, 2003.

[33] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM*, 1998, pp. 1240–1247.

[34] M. Waldogel, G. Varghese, J. Turner, and B. Plattne, "Scalable high speed IP routing lookups," in *Proc. ACM SIGCOMM*, vol. 27, pp.25–36, 1997.

[35] SSFNet toolkit. [Online]. Available: http://www.ssfnet.org

Lai Mingche received Ph.D. degree in computer engineering from National University of Defense Technology, China, in 2008. From 2008 to 2010, he was a Lecturer with the Department of Computer Science. Since 2007, he was also a Faculty Member with National Key Laboratory for Parallel and Distributed Processing of China. His research interests include network architecture, hardware/software codesign, and optical communication.



Gao Lei received the Ph.D. degree from National University of Defense Technology, Hunan, China, in 2009. She is currently an Assistant Researcher at National Key Laboratory for Parallel and Distributed Processing of China. Also she is currently worked together with Intel, served as a System Design Engineer. Her research interests include BGP protocol, multi-thread programming, computer-aided design, and hardware/software Co-design.