Relaxed Min-Augmented Range Trees for Dynamic IP Router Tables

†Technical Report – Albert-Ludwigs-Universität Freiburg

Christine Maindorfer

Institut für Informatik†. Lehrstuhl Algorithmen & DatenStrukturen.
maindorfer@informatik.uni-freiburg.de

April 10, 2007

Preamble. When an Internet router receives a packet p from an input link interface, it uses p's destination field to look up a routing database. The result of the lookup provides an output link interface, to which packet p is forwarded. Ever increasing routing table sizes, traffic volume and link speeds demand efficient lookup algorithms. In router tables, IP lookup is intermixed with updates, hence it is important to avoid extensive locking. Furthermore, updates should be processed as fast as possible in order to minimize packet loss and packet delay.
In relaxed balanced data structures, rebalancing is uncoupled from the update such that the rebalancing tasks can be performed gradually after urgent updates.
We will investigate how the min-augmented range tree, a structure which can be utilized for IP lookup, can be relaxed.

1 Introduction

Internet routing operates at the level of prefixes. Rather than having each router store a database entry for all possible destination IP addresses, the router stores address prefixes that represent a group of addresses reachable through the same interface. From a routing point of view, the Internet can be considered to be partitioned into a number of independent sub-networks, the so-called Autonomous Systems (ASes). Routers in different ASes use the external Border Gateway Protocol (*EBGP*) to exchange network reachability information. After applying local policies, a BGP router selects a single best route and advertises it to other BGP routers within the same AS through internal BGP(*IBGP*) sessions. A router sends an announcement to notify its neighbors of a new route to the destination prefix and sends a withdrawal to revoke the route when it is no longer available. From these updates an IP router constructs a forwarding table that tells the router to which interface an incoming packet with a certain destination prefix has to be sent. If several prefixes match an incoming packet, ties are broken by selecting the longest matching prefix for that destination address. This tie-breaker is refered to as the most-specific tie-breaking rule.
An update message can advertise, at most, one set of path attributes, but multiple destinations, provided that the destinations share these attributes. Furthermore, an update message can list multiple routes that are to be withdrawn from service. Using this capability of comprising several announcements and withdrawals into one update message is highly recommended sine internet traffic will be

† Fakultät für Angerwandte Wissenschaften. Georges-Köhler-Allee 51. D-79110, Freiburg, Deutschland.

reduced.
A routing change might trigger multiple transient routes during the route convergence process before the final stable route is selected. In order to control the rate of BGP messages pushed into the network, each router uses a rate-limiting timer, called the Minimum Route Advertisement Interval (MRAI) timer. The MRAI determines the minimum amount of time that must elapse between routing updates to a particular destination for the same neighbor. If new routes are selected multiple times while awaiting the expiration of MRAI, the last route selected shall be advertised at the end of MRAI. In the BGP specification [1] the recommended value for the MRAI timer on EBGP connections is 30 seconds and 5 seconds on IBGP connections.

The deployment of interactive services such as VoIP on the Internet demands the network to maintain good end-to-end performance. Experimental results [2] show a strong correlation between VoIP intelligibility and BGP updates. The dynamics of the convergence process is tied closely to the interaction of the message handling process and the MRAI timer and also the topology of the network. Deshpande and Sikdar [3] simulation results show that an adaptive MRAI timer significantly reduces convergence times, yet increases the advertised update messages.
Shand and Bryant [4] suggest to store not only the best path but also the second best at each router in order to minimize the impact of routing failures during updates. This can potentially reduce the chance that a router loses routes at the same time.

1.1 Previous Work

A multitude of solutions for the representation of *static* as well as *dynamic* rule tables have been proposed. Static data structures are created for an input data set with infrequent updates. Typically in 'static' data structures, insertions and deletions are batched and the data structure is reconstructed from scratch as needed. Dynamic structures allow for efficient update operations.
Many of the proposed data structures are based on the binary trie structure, e.g. [5] [6] [7]. Lu and Sahni [8] propose a method to partition a static IP router table such that each partition is represented using a base structure such as a multibit trie [9] or a hybrid shape shifting trie [10].
Lee *et al.* [11] propose an algorithm which is based on the segment tree. Given n IP prefixes, their algorithm performs IP address lookup in $O(\log n)$ time. It also handles insertion and deletion of IP prefixes efficiently.
Lu and Sahni [12] develop an enhanced interval tree LMPBOB (longest matching prefix binary tree on binary tree) which permits lookup in $O(W)$ time, where W is the length of the longest prefix, and filter insertion and deletion in $O(\log n)$ time each.
Warkhede, Suri and Varghese [13] introduce an IP lookup scheme based on a multiway range tree with worst-case search and update time of $O(\log n)$, where n is the number of prefixes in the forwarding table.
Lu and Sahni [14] show that each of the three operations insert, delete and longest matching prefix search may be performed in $O(\log n)$ time in the worst case using a priority-search tree.

1.2 Relaxed Balancing

In relaxed balanced data structures, the rebalancing is uncoupled from the update and may be arbitrarily delayed and interleaved. This contrasts with strict balancing, where rebalancing is performed immediately after an update. Trees with relaxed balance are defined to facilitate fast updating in a concurrent database environment, since the rebalancing tasks can be performed gradually after urgent updates. Updating and rebalancing are uncoupled such that extensive locking can be avoided in connection with updates. However, weaker constraints than the usual ones are maintained such that the tree can still be balanced efficiently independent of the updating processes. The choice of constraints and conditions under which the different rebalancing operations can be carried out

is critical for the overall complexity. Uncoupling was first discussed in connection with red-black trees [15], and later in connection with AVL trees [16]. Since then, several relaxed balancing schemes have been proposed [17] [18] [19]. Relaxed data structures with group updates have been proposed in [20] [21] [22].
Hanke [23] presents an experimental comparison of the strictly balanced red-black tree and the chromatic tree, a relaxed balanced red-black tree. The results indicate that the chromatic tree has a significant advantage over the strictly balanced red-black tree.
In router tables, IP lookup is intermixed with updates, hence it is important to avoid extensive locking. Furthermore, updates should be processed as fast as possible in order to minimize packet loss and to support higher update message rates for faster convergence times.
We suggest an elegant way for the representation of dynamic routing tables, namely relaxed balanced min-augmented range trees, which support updates in $O(\log n)$ time. To our knowledge, there are none other approches which examine relaxed balancing for structures utilized for IP lookup.

2 Geometric Interpretation of IP lookup

A prefix $b_1 \dots b_k*$ can be mapped to an interval in the form of $[b_1 \dots b_k 0 \dots 0, b_1 \dots b_k 1 \dots 1]$. For example, if the prefix length is limited by 5, $0010*$ is represented by $[4, 5]$. An incoming packet with destination address $d = b_1, \dots, b_w$ can be mapped to a point $p \in U$, where $U = [0, 2^w - 1]$ and $w = 32$ for IPv4 and $w = 128$ for IPv6.
In geometric terms, IP lookup corresponds to performing a stabbing query for a given set of intervals and a given query point. The most-specific tie-breaking rule selects the longest matching prefix $r \in R$ among all filters that match an incoming packet p. The longest matching prefix corresponds to the *most-specific* interval of all intervals stabbed by a query point. Filter f_1 is more specific than filter f_2 iff $f_1 \subset f_2$. If two filters intersect, neither is more specific than the other. A set of prefix intervals has the property that any two intervals are either disjoint or one is completely contained in the other. We refer to these sets as nested sets. Hence, for each query point d, there is a unique defined most-specific interval stabbed by d.

Intervals on the line can be mapped to points in the plane and vice versa, because both entities are defined by two values. If we map an interval (l, r) with start point l and finish point r on the line to the point (r, l) in the plane, a set of intervals on the line is mapped to a set of points below the main diagonal in the plane. Let us denote this mapping by *map1*, following the notation of Lu and Sahni [14]. It has been observed that stabbing queries for sets of intervals on the line can be translated to range queries for so called south-grounded, semi-infinite ranges of points in the plane [24]. More precisely: $d \in [l, r]$ if *map1* (l, r) is to the right and below point (d, d).

For two ranges (l, r) and (l', r') the point *map1* (l, r) lies to the left and above the point *map1* (l', r') iff the range (l, r) is contained in the range (l', r').

Let us now assume that the given set R of ranges is a prefix set. Then, for each d the most specific range in R containing d, *msr(d)*, is well defined. Hence, in the corresponding set $S =$ *map1(R)* of points in the plane representing the set R of ranges there is a uniquely defined topmost-leftmost point in the semi-infinite range to the right and below the query point $p = (d, d)$ on the main diagonal. Thus, solving the dynamic version of the IP lookup problem for prefix filters means to maintain a set of points in the plane for which we can carry out insertions and deletions of points and answer topmost-leftmost-queries efficiently. Topmost-leftmost-queries can be reduced to leftmost-queries when ensuring that no two points have the same x-coordinate. This can be accomplished by mapping each point (x, y) to the point $(2^w x - y + 2^w - 1, y)$ [14]. Note that it is not possible to have a leftmost point which is not also the highest point in the semi-infinite range to the right and below a query point on the main diagonal. This is due to the fact that prefix sets are nested.

3 Min-Augmented Range Trees

A min-augmented range tree *(MART)* [25] storing a set of points with pairwise different x-coordinates stores the points at the leaves such that it is a leaf-search tree for the x-coordinates of points. The internal nodes have two fields, a router field guiding the search to the leaves and a min-field. In the router field we store the maximum x-coordinate of the left subtree, and in the min-field we store the minimum y-coordinate of any point stored in the leaves of the subtrees of the node.

Consider the following example: $S = \{(1,3),(2,4),(3,7),(4,2),(5,1),(6,6),(7,5),(8,4)\}$ of eight points. A min-augmented range tree storing this set of eight points is visualized in Fig. 1.

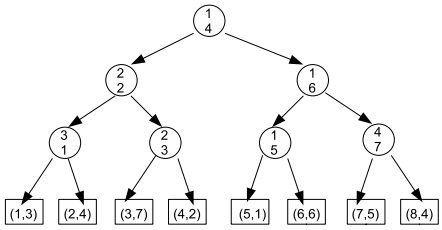

Figure 1. Example of a min-augmented range tree.

First we show how to answer a leftmost-, or *MinXinRectangle* $(x_{left}, \infty, y_{top})$ query, that is, to find the point p with minimal x-coordinate in the semi-infinite range $x \geq x_{left}$ and with y-coordinate below the threshold value y_{top}. In order to find this point, we first carry out a search for the boundary value x_{left}. It ends at a leaf storing a point with minimal x-coordinate larger than or equal to x_{left}. If this point has a y-coordinate below the threshold value y_{top}, we are done. Otherwise we retrace the search path for x_{left} bottom-up and inspect the roots of subtrees falling completely into the semi-infinite x-range. These roots appear as right children of nodes on the search path. Among them we determine the first one from below (which is also the leftmost one and) which has a min-field value below the threshold y_{top}. This subtree must contain the answer to the *MinXinRectangle* $(x_{left}, \infty, y_{top})$ query stored at its leaf. In order to find it, we recursively proceed to the left child of the current node, if its min-field shows that the subtree contains a legal point, i.e. if its min-field is (still) below the threshold, and we proceed to the right child only, if we cannot go to the left child (because the min-field of the left child is above the threshold y_{top}). We can find the desired point in time which is proportional to the height of the underlying leaf-search tree. Hence it is desirable to maintain the underlying leaf-search tree balanced. All we have to show is that the augmented information stored in the min-fields of nodes can efficiently be maintained if we carry out an update operation and rebalance the underlying search tree. In order to show this, it is appropriate to think of an update operation for the underlying balanced leaf-search tree as consisting of two successive phases. In the first phase, we insert or delete a point as in a normal (unbalanced) binary leaf-search tree, and in the second phase we retrace the search path and carry out rebalancing operations, if necessary. In order to update the information stored in the min-fields of internal nodes, the first phase has to be extended as follows. We retrace the search path and carry out a tournament starting from the leaf affected by the update operation: We recursively consider the min-fields of the current node and its sibling and store the minimum of both in the min-field of their common parent. In this way we correctly update the information stored in the min-fields after the first phase. Instead of retracing the search path in order to update the min-fields, the insertion process could also modify the min-fields top-bottom. In order to show that this information can also be maintained during the second phase, i.e. during rebalancing, let us consider a right rotation. Here we assume that a, b, c, d, e

are the routers in increasing x-order stored in the router fields of the internal nodes. The values of the min-fields before the rotations are u, v, w, x, y and u, v', w, x', y after the rotation. Note that u, w, y have not to be changed, because their subtrees are not affected by the rotation. We have just to update the min-fields of the nodes A and B. Note, however, that the min-value stored at node A is (still) the overall min-value x of all subtrees 1, 2, and 3, hence, we define $x' = x$. Choosing $v' = min(w, y)$ will finally restore the min-fields correctly, cf. Figure 2.

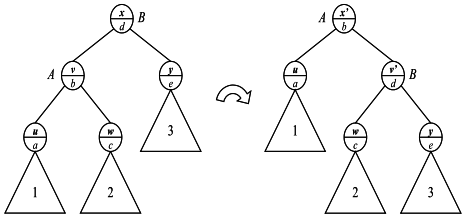

Figure 2. MART right rotation. Min-field values are shown on top of the router values.

Rotations and the process of maintaining the augmented min-information become strictly local, hence we can freely choose an underlying balancing scheme for min-augmented range trees. Furthermore, the locality property enables us to decouple the update and rebalancing operations as will be shown in Section 5.

4 Relaxed Balancing

Instead of requiring that the balance condition is restored immediately after each update, the balance conditions are relaxed such that the rebalancing operations can be delayed and interleaved with search and update operations. Several relaxed balancing schemes have been proposed, especially for red-black trees [26] [18]. In the following we will present the relaxed red-black tree as proposed in Hanke, Ottmann and Soisalon-Soininen [18] as we utilized this scheme to relax the min-augmented range tree. This scheme can be applied to any class of balanced trees. The main idea is to use the same rebalancing operations as for the standard version of the tree. In the following we shortly recap the standard red-black tree.

4.1 Standard Red-Black Tree

A red-black tree is a binary search tree with the following red-black properties:

1. Every node is either red or black.
2. Every leaf (NULL) is black.
3. If a node is red, then both its children are black.
4. Every path from the root to a leaf contains the same number of black nodes.
5. The root node is black.

These constraints enforce a critical property of red-black trees: that the longest possible path from the root to a leaf is no more than twice as long as the shortest possible path. A red-black tree with n internal nodes has height at most $2\lg(n+1)$. The immediate result of an insertion or removal may violate the properties of a red-black tree. Restoring the red-black properties requires a small number ($O(\log n)$ or amortized $O(1)$) of color changes and no more than three tree rotations (only two for insertion).

Insertions

In order to insert a new key we first locate its position among the leaves and replace the leaf by an internal red node p with two black leaves. If the parent of p is red, we must restore the balance condition and call the rebalancing procedure *up-in* for p, cf. Figure 3.

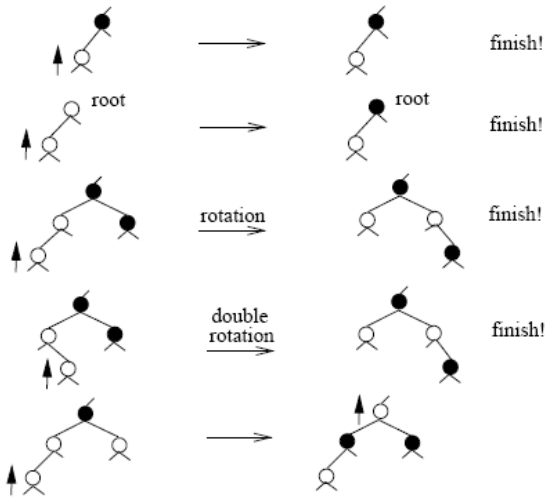

Figure 3. Call of the rebalancing procedure *up-in* (denoted by ↑). Filled nodes denote black nodes. From [18].

Deletions

In order to delete a key we first locate its position among the leaves and then remove the leaf together with its parent. If the parent is red, then remove the leaf together with its parent. If the removed parent was black the balance condition is violated. If the sibling is red, we just change it to black. Otherwise, the removal leads to a call of the rebalancig procedure *up-out* for the remaining leaf, cf. Figure 4.

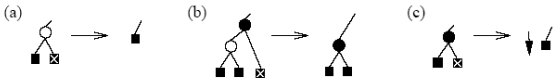

Figure 4. Deletion of an item (denoted by x) and call of the rebalancing procedure *up-out* (denoted by ↓). From [18].

The task of the procedure *up-out* attached to some node p is to increase the black height of the subtree rooted at p by one. It either performs a structural change and settles the request or it moves up in the tree, cf. Figure 5.

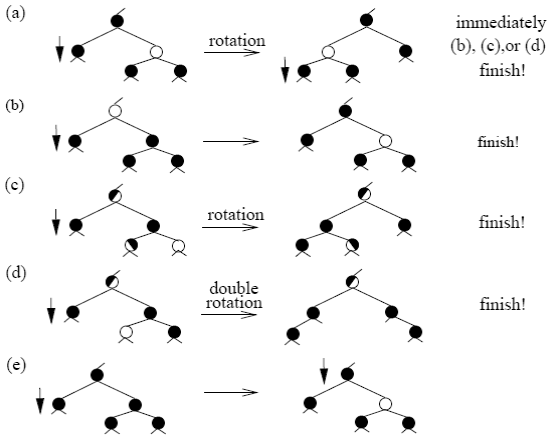

Figure 5. Call of the rebalancing procedure *up-out* (denoted by ↓). Half filled nodes denote nodes that are either black or red. From [18].

4.2 Relaxed Balanced Red-Black Tree

As we want to uncouple the rebalancing from an update, we only **deposit** an *up-in* or an *up-out* request instead of calling these procedures immediately after an update.

The relaxed balance conditions require that

1. on each path from the root to the leaf, the sum of the number of black nodes plus the number of *up-out* requests is the same
2. each red node has either a black parent or an *up-in* request
3. all leaves are black

The rebalancing requests can be carried out concurrently as long as they do not interfere at the same nodes. This can be achieved by settling the rebalancing requests in a top-down manner.

If the deletion falls into a leaf with a red parent, and the leaf's sibling has an *up-out*, *up-in*- or removal request, it does not interfere with the deletion and remains attached, and the leaf together with its parent is removed, as in Figure 4 (a). Otherwise, we just deposit a removal request at the appropriate leaf. The deallocation is thus part of the rebalancing process. To handle such a removal request is only allowed if the relaxed balancing condition is not violated. If the removal request falls into a leaf with an *up-out* request or whose parent has an *up-out* request, these requests have to be settled first.

In order to facilitate the locating of rebalancing requests in the tree, we utilize a problem queue as proposed in [27]. For each type of request, we maintain a seperate queue. If all problem queues are empty, the tree is (strictly) balanced. To avoid side effects, only one element in a problem queue may link to a node. Each node maintains an additional bit for each problem queue, which is set as soon as a link to that node is inserted in the queue. When the request in the queue is deleted, the rebalancing process resets this bit to zero.

Since the same rebalancing operations as for the standard version of the tree are used, the number of rebalancing operations from the strict balancing scheme carry over to relaxed balancing.

Interleaving updates

If an insertion falls into a leaf which has a removal request and/or an *up-out* request then if it has a removal request abandon the removal request and reinsert the key at that leaf. If the leaf has an *up-out* request remove the *up-out* request and replace the leaf by an internal black node with two leaves. If a deletion falls into a leaf which has a red parent with an *up-in* request the leaf is immediately deleted and the *up-in* request abandoned, cf. Figure 6.

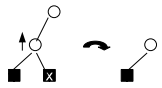

Figure 6. Deletion of an item (denoted by x).

Concurrent handling of rebalancing transformations

Two *up-out* requests that occur at sibling nodes are in conflict. However, this can be solved by applying the transformations in Figure 7 (a) or (b). If two *up-out* requests occur in the same area as in Figure 7 (c), this conflict can be settled by one rotation and recoloring.

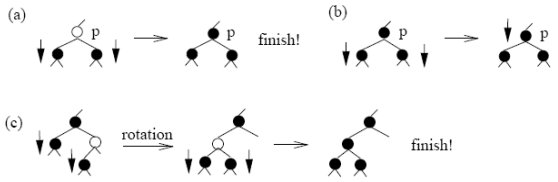

Figure 7. Concurrent handling of the procedure *up-out* (denoted by ↓). From [18].

5 Relaxed Min-Augmented Range Trees

We will now examine how the min-augmented range tree and the relaxed balanced red-black tree can be combined into the relaxed min-augmented range tree RMART. The insertion process modifies the y-fields top-bottom. In order to update the min-fields after a removal we introduce the update request *up-y*. Hence, in addition to *up-in*, *up-out* and removal requests, red and black nodes can also have *up-y* requests. Furthermore, we need one additional problem queue for the *up-y* requests. If a node p has an *up-y* request we must update the predecessors' min-fields if necessary. To settle an *up-y* request we compare the min-field values of p and it's sibling, store the minimum m of these values in *parent(p)*, delete the *up-y* request at p and shift the *up-y* request to *parent(p)*, if it doesn't have an *up-y* request yet. If m equals the min-field value of *parent(p)*, we just delete the *up-y* request at p.
All features of the min-augmented range tree remain valid, except for the x-value property. Every node may have an x-value that is larger or equal the highest x-value of its left subtree and smaller than the smallest x-value in the right subtree. Hence, we do not have to update the x-values after a deletion.

The relaxed balance conditions stay unmodified:

1. on each path from the root to the leaf, the sum of the number of black nodes plus the number of *up-out* requests is the same
2. each red node has either a black parent or an *up-in* request
3. all leaves are black

If a deletion falls into a leaf with a red parent, remove the leaf together with its parent, cf. Figure 4 (a). Additionally, attach an *up-y* request to the leaf's sibling. Otherwise, as in the case of relaxed balanced red-black trees, we only deposit a removal request. When we settle a removal request which resides in a leaf with a black parent and red sibling, we remove the leaf together with its parent, and color the sibling black (as in Figure 4 (b)). Additionally, we attach an *up-y* request to the leaf's sibling. When we settle a removal request which resides in a leaf with a black parent and sibling, we remove the leaf together with its parent, and attach an *up-out* request to the leaf's sibling (Figure 4 (c)). Additionally, we attach an *up-y* request to the leaf's sibling.

5.1 Interleaving updates

Every node has only one request of the same type. If an insertion falls into a leaf with an *up-y* request, the procedure stays the same as for the relaxed balanced red-black tree and the *up-y* request is shifted to the new internal node. If this leaf has a removal and an *up-y* request, then we insert the new point and detach both requests.

If the deletion falls into a leaf whose red parent has an *up-in* request, the same transformation as in Figure 6 is applied. Additionally, we have to attach an *up-y* request to the leaf's sibling, cf. Figure 8.

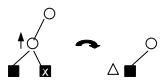

Figure 8. Deletion of an item and attachment of the rebalancing procedure *up-y* (denoted by △).

If an *up-out* request and an *up-y* request meet at a node p, then the *up-out* transformations that don't involve rotations do not interfere with the *up-y* requests, cf. Figure 5 (b) and (e). If rotations are involved, the *up-y* request at p or at nodes that are involved in the rotation are shifted out of the sphere of influence of the rotation prior to the *up-out* transformations. As in the case of rotations in standard min-augmented range trees, the y-values of the involved nodes are maintained during a rotation.

5.2 MinXinRectangle search

Definition 1. *Let v be a leaf with min-field y. The* y-min path *contains v and all ancestor nodes of v with min-field y.*

The length of a y-min path is the number of internal nodes it contains. The maximum length of a y-min path equals the height of the tree. A node k belongs to the y-min path of a leaf v, if v stores the minimum y-value of all y-values of leaves stored in the subtrees of k.

Theorem 1. *The average y-min path length is* 2.

Proof. Let N be the number of internal nodes. The sum of the y-min path lengths is N, as each node belongs to exactly one y-min path. Every leaf plays a turnament with its sibling and the winner contributes to the sum of the path lengths. Hence, the average y-min path length is $2N/(N+1)$.

If the *MinXinRectangle*-search ends at a leaf with a higher y-value than the threshold, the search has to track back. If the y-fields along the path have not yet been updated, i.e. if *up-y* requests are encountered or the leaf where the search ends is marked with a removal request, the search might follow a path which again ends at a leaf with an invalid y-value, i.e. y is greater than the threshold, or a removal request respectively. Hence, the search has to be modified in such a way that all leaves with a valid x-value are visited one after the other (in ascending x-values) until a leaf with a valid y-value has been found. In the worst case, all y-values are invalid, giving a runtime of $O(n)$ in a tree with n inner nodes. Hence, *up-y-* and removal requests should be handled frequently, in order to avoid long search paths. However, only two nodes on average have to be updated in order to settle an *up-y* request, which results in a fast handling of the *up-y* requests. It also benefits the search for the most specific range, as backtracking is limited by two internal nodes on average.

6 Concurrency control

A mechanism for concurrency control in strictly balanced search trees is the locking protocol by Ellis [28]. Nurmi and Soisalon-Soininen [29] propose a modification of Ellis' scheme for relaxed balanced search trees. Both protocols use r,- w- and x-locks. Several nodes can r-lock a node at the same time. Only one process can w- or x-lock a node. Furthermore, a node can be both w-locked by one process and r-locked by several other processes. An x-locked node cannot be r- w- or x-locked by another process. An update process uses w-locks, if it wants to exclude other update processes, but does not want to exclude search processes. If an update process changes the search structure, then it uses x-locks.
A w-lock can be changed into a x-lock if the node is not r-locked by another process. A x-lock can always be changed into a w-lock and a w-lock into a r-lock.
Lock coupling means that a process locks the node to be visited next before it releases the lock on the currently visited node.
Search processes use r-lock coupling.

6.1 Strict balanced Trees

In strict balanced trees, an insertion process w-locks the entire path from the root to the leaf, where the key is inserted, in order to be able to rebalance immediately after the insertion. If the key is already in the tree, all locks along the path are released and the insertion process terminates. Otherwise, the leaf's w-lock is changed into a x-lock. The leaf is changed into an inner node with two leaves and the key is inserted. Afterwards, the tree is rebalanced. To check which rebalancing transformation applies, all relevant nodes are w-locked . If only the node colors change, w-locks are sufficient. If rotations are involved, then the w-locks are changed into x-locks in top-down direction. After the rotaion, the locks of the nodes that are locally in balance are released. When the tree is completely in balance, all remaining w-locks are released and the insertion process terminates.
The locking strategy of a deletion process is similar. The difference is that not only the leaf, but also the father and the sibling have to be x-locked. This is done in top-down direction after they have been w-locked. The content of the sibling is copied into the father node, and the two leaves are deleted. The locking strategy for rebalancing the tree remains the same as in the insertion process.

6.2 Relaxed balanced Trees

In relaxed balancing, it is sufficient to use w-lock coupling during the search phase of an update operation. Once arrived at a leaf, the w-lock is changed into a x-lock. In case of an insertion, the

leaf is changed into an internal node with two leafs. Then, the node is unlocked and the insertion process terminates.
Generally, the deletion process just attaches a removal request and the actual deletion is part of the rebalancing process. Analog to the insertion process, the leaf is x-locked and the removal request attached before the x-lock is released.
A rebalancing process w-locks the nodes while checking whether a transformation applies. If a transformation applies, it changes all w-locks to x-locks. Nurmi and Soisalon-Soininen suggest to traverse the tree nondeterministically in order to locate the rebalancing requests. This implies that the top-most node can be locked first among all nodes that have to be considered, hence the locking scheme is deadlock free. If a problem queue is used to locate the requests, the scheme must be extended in an appropriate way, cf. Hanke [23]: when a rebalancing process tries to w-lock its parent that is w- or x-locked by another process, the rebalancer immediately releases all locks it holds. The rebalancing processes are the only processes that also w-lock bottom-up. Since they immediately release all locks if a requested lock cannot be granted, deadlock situations are eliminated. The rebalancing processes basically all use the same locking strategy and are not privileged, i.e. if a lock cannot be granted, the rebalancer immediately releases all locks it holds. If a request pointing at node p is taken out of a queue, then the rebalancing process tries to w-lock the parent and if necessary the grandfather. If it is not possible to w-lock these nodes, i.e. if they are w -or x-locked by another process, the locks are immediately released bottom-up. If the topmost node, that is relevant for the transformation is successfully locked, then all other relevant nodes are w-locked top-bottom. Along the way, we check if a transformation can be applied. If no further node can be locked and no transformation can be applied, then all locks are released. If a transformation can be applied, then the w-locks are changed to x-locks top-bottom. If a x-lock cannot be granted, all locks are released. Then, the rebalancing process either tries to lock the appropriate nodes again, or it appends the request to the corresponding problem queue. Otherwise, if all x-locks have been granted, the transformation is applied, and if necessary, a newly generated rebalancing request is appended to the respective queue, and all locks are released.

In the following we extend the locking scheme by Hanke such that it can be applied for the RMART.

Search Process
The *MinXinRectangle* search uses r-lock coupling when it tracks back.

Update Process
Since the insertion process modifies the y-fields top-bottom, it uses x-locks in addition to w-lock coupling. It tries to get the lock until it receives it. It first w-locks and then x-locks a node p. After the y-field has been updated, the x-lock is changed into a w-lock. Then, it tries to w-lock the appropriate son. If this is successful, it releases p's lock and x-locks the son. Once arrived at a leaf, it inserts the new item and then releases the x-lock.

The proposed extension of the locking protocol from [23] is deadlock-free, just as the underlying protocol. The search process additionally traverses the tree bottom-up. Since it uses only r-locks, it does not introduce deadlock situations.

7 Work in Progress

Currently we are in the process of implementing and designing the test environment to benchmark the RMART with the concurrent standard version of the tree. The RMART has been implemented in [30]. To conduct our experiments, we will use real and up-to-date routing data that is supplied

by the Advanced Network Technology Center of the University of Oregon within the framework of the Route Views Project [31].

References

1. Y. Rekhter, T. Li, and S. Hares. (2006, jan) A border gateway protocol 4 (bgp-4). Request for Comments: 4271. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4271.txt
2. N. Kushman, S. Kandula, and D. Katabi, "Can you hear me now?!: it must be bgp," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 75–84, 2007.
3. S. Deshpande and B. Sikdar, "On the impact of route processing and mrai timers on bgp convergence times," in *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, vol. 2, 2004, pp. 1147– 1151.
4. M. Shand and S. Bryant. (2006, oct) Ip fast-reroute framework. Internet Draft. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtgwg-ipfrr-framework-06.txt
5. K. S. Kim and S. Sahni, "Efficient construction of pipelined multibit-trie router-tables," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 32–43, 2007.
6. H. Song, J. Turner, and J. Lockwood, "Shape shifting tries for faster ip route lookup," in *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols (ICNP'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 358–367.
7. I. Ioannidis, A. Grama, and M. Atallah, "Adaptive data structures for ip lookups," *J. Exp. Algorithmics*, vol. 10, p. 1.1, 2005.
8. H. Lu and S. Sahni, "Recursively partitioned static ip router-tables," in *IEEE Symposium on Computers and Communications*, 2007.
9. V. Srinivasan and G. Varghese, "Faster ip lookups using controlled prefix expansion," in *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1998, pp. 1–10.
10. W. Lu and S. Sahni, "Succinct representation of static packet classifiers," in *IEEE Symposium on Computers and Communications*, 2007.
11. I. Lee, K. Park, Y. Choi, and S. K. Chung, "A simple and scalable algorithm for the IP address lookup problem," *Fundamenta Informaticae*, vol. 56, no. 1,2, pp. 181–190, 2003.
12. H. Lu and S. Sahni, "Enhanced interval trees for dynamic IP router-tables," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1615–1628, 2004.
13. P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: scalable IP lookup with fast updates," *Computer Networks*, vol. 44, no. 3, pp. 289–303, 2004.
14. H. Lu and S. Sahni, "O(log n) dynamic router-tables for prefixes and ranges," *IEEE Transanctions on Computers*, vol. 53, no. 10, pp. 1217–1230, 2004.
15. L. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, 1978, pp. 8–21.
16. J. L. W. Kessels, "On-the-fly optimization of data structures," *Commun. ACM*, vol. 26, no. 11, pp. 895–901, 1983.
17. Larsen and Fagerberg, "B-trees with relaxed balance," in *IPPS: 9th International Parallel Processing Symposium*. IEEE Computer Society Press, 1995. [Online]. Available: citeseer.ist.psu.edu/larsen95btrees.html
18. S. Hanke, T. Ottmann, and E. Soisalon-Soininen, "Relaxed balanced red-black trees," in *CIAC '97: Proceedings of the Third Italian Conference on Algorithms and Complexity*. London, UK: Springer Verlag, 1997, pp. 193–204. [Online]. Available: citeseer.ist.psu.edu/hanke97relaxed.html
19. K. S. Larsen, T. Ottmann, and E. Soisalon-Soininen, "Relaxed balance for search trees with local rebalancing," *Acta Informatica*, vol. 37, no. 10, pp. 743–763, 2001. [Online]. Available: citeseer.ist.psu.edu/larsen97relaxed.html
20. K. S. Larsen, "Relaxed red-black trees with group updates," *Acta Informatica*, vol. 38, no. 8, pp. 565–586, 2002.
21. L. Malmi and E. Soisalon-Soininen, "Group updates for relaxed height-balanced trees," in *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM Press, 1999, pp. 358–367.

22. K. S. Larsen, "Relaxed multi-way trees with group updates," in *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM Press, 2001, pp. 93–101.
23. S. Hanke, "The performance of concurrent red-black tree algorithms," *Lecture Notes in Computer Science*, vol. 1668, 1999. [Online]. Available: citeseer.ist.psu.edu/hanke98performance.html
24. E. M. McCreight, "Priority search trees." *SIAM J. Comput.*, vol. 14, no. 2, pp. 257–276, 1985.
25. T. Lauer, T. Ottmann, and A. Datta, "Update-efficient data structures for dynamic IP router tables," *International Journal of Foundations of Computer Science*, vol. 18, no. 1, pp. 139–161, 2007.
26. T. Ottmann and E. Soisalon-Soininen, "Relaxed balancing made simple, Tech. Rep. 71, Jan. 1995. [Online]. Available: citeseer.ist.psu.edu/ottmann95relaxed.html
27. J. Boyar and K. S. Larsen, "Efficient rebalancing of chromatic search trees," in *Proceedings of the 30th IEEE symposium on Foundations of computer science*. Orlando, FL, USA: Academic Press, Inc., 1994, pp. 667–682.
28. C. S. Ellis, "Concurrent search and insertion in avl trees." *IEEE Trans. Computers*, vol. 29, no. 9, pp. 811–817, 1980.
29. O. Nurmi and E. Soisalon-Soininen, "Chromatic binary search trees. a structure for concurrent rebalancing." *Acta Inf.*, vol. 33, no. 6, pp. 547–557, 1996.
30. T. Seddig, "Balancierung von Datenstrukturen zur Lösung des Paketklassifizierung-Problems," Diplomarbeit, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Apr. 2006.
31. U. of Oregon. University of oregon route views project. [Online]. Available: http://www.routeviews.org/