The Performance of Concurrent Red-Black Tree Algorithms

Institut für Informatik — Report 115

Sabine Hanke
Institut für Informatik, Universität Freiburg
Am Flughafen 17, 79110 Freiburg, Germany
Email: hanke@informatik.uni-freiburg.de

November 1998

Abstract

Relaxed balancing has become a commonly used concept in the design of concurrent search tree algorithms. The idea of relaxed balancing is to uncouple the rebalancing from the updating in order to speed up the update operations and to allow a high degree of concurrency. Many different relaxed balancing algorithms have been proposed, especially for red-black trees and AVL trees, but their performance in concurrent environments is not yet well understood.

This paper presents an experimental comparison of three relaxed balancing algorithms for red-black trees. Using the simulation of a multi processor environment we study the performance of chromatic trees, the algorithm that is got by applying the general method of how to make strict balancing schemes relaxed to red-black trees, and the relaxed red-black tree. Furthermore, we compare the relaxed balancing algorithms with the standard red-black tree, i.e. the strictly balanced red-black tree combined with the locking scheme of Ellis. We find that in a concurrent environment the relaxed balancing algorithms have a significant advantage over the strictly balanced red-black tree. With regard to the average response time of the dictionary operations the three relaxed balancing algorithms perform almost identically. Regarding the quality of the rebalancing we find that the chromatic tree has the best performance.

1 Introduction

A *dictionary* is a scheme for storing a set of data such that the operations *search*, *insert*, and *delete* can be carried out efficiently. Standard implementations of dictionaries are balanced search trees like red-black trees, AVL-trees, or B-trees. In order to speed up the response time of the dictionary operations and to allow a high degree of concurrency, the idea of *relaxed balancing* was introduced. Instead of requiring that the balance condition is restored immediately after each update operation in the search tree, the actual rebalancing transformations are uncoupled from the update operations and may be arbitrarily delayed and interleaved.

Relaxed Balancing was first suggested in [9] for red-black trees. The first actual solution for relaxed balancing is presented by Kessels [14]. The trees in Kessel's solution are AVL-trees and the only allowed updates are insertions. An extension to the general case where deletions are also allowed is presented by Nurmi et al. [28]. This solution is analyzed in [16], and it is shown that for each update operation in a tree with maximum size n, $O(\log n)$ new rebalancing operations are needed.

Relaxed balancing for B-trees is introduced in [28] and further analyzed in [18]. Another concurrent B-tree algorithm that uses the concept of relaxed balancing is the B^{link}-tree [15, 21, 30].

Nurmi and Soisalon-Soininen [26, 27] propose a relaxed version of red-black trees which they call a *chromatic tree*. Chromatic trees are analyzed by Boyar and Larsen [6], and a more efficient set of rebalancing transformations is defined. It is shown that the number of rebalancing operations per update is $O(\log(n+i))$, if i insertions are performed on a tree which initially contains n leaves and that the number of rebalancing transformations per update that perform rotations is constant. Boyar et al. prove for a slightly modified set of rebalancing operations that starting with an empty chromatic tree only an amortized constant amount of rebalancing is necessary after an update [5].

A *general method* of how to make known rebalancing algorithms relaxed in an efficient way is proposed in [11]. With an example of red-black trees it is shown that essentially the same set of rebalancing transformations as used in the strict case [31] can also be used for the relaxed case, and that the number of needed rebalancing operations known from the strict balancing scheme carry over to relaxed balancing. In [29] this method is applied to stratified trees. Furthermore, in [19] the technique of [11] is described in an abstract way for the class of all search trees with local bottom-up rebalancing.

Relaxed red-black trees, another version of a relaxed balanced red-black tree that is based on the rebalancing algorithms for the strict case [31], are proposed by Larsen [17]. The difference to the previous solutions is that large rebalancing transformations are split into smaller ones consisting only of a single or double rotation. It is shown that the rebalancing is also amortized constant.

Bougé et al. present a relaxed balancing scheme for AVL-trees that rebalance any binary search tree with n nodes in $O(n)$ steps [3, 4]. The algorithm is highly fault tolerant and can also be combined with other rebalancing schemes.

Soisalon-Soininen and Widmayer introduce *height-valued binary search trees*, a relaxed version of AVL-trees which is based on the standard balancing transformations for AVL trees [32]. This solution is analyzed in [20], and it is proved that the number of rebalancing operations needed to restore the AVL balance condition is bounded by $O(M \log(M + N))$, where M is the number of updates to an AVL tree of initial size N. Height-valued trees differ from all previous solutions for relaxed balancing in the important aspect that a rotation never is performed if the underlying search tree happens to fulfill the balance condition before any rebalancing has been done.

The *rank valued tree* proposed by Malmi is a close relative of the height-valued trees and implements a similar idea to symmetric balanced B-trees [24].

An analogous algorithm for red-black trees is proposed in [10]. It is shown that each performed structural change indeed correspond to a real imbalance situation in the tree, if the handling of overweight conflicts is forced to happen bottom-up. Furthermore, as it is mentioned in passing, the weight-balancing transformations can also be used to handle consecutive red nodes, if a logarithmic number of rotations per insertion is not minded. The idea is to transform red-red conflicts into analogous problems as the balance conflicts caused by deletions by coloring red nodes black and using negative units of overweight. Afterwards, the differences between the overweights of sibling nodes can be evened out by performing one of the weight-balancing transformations.

An idea very similar to the negative units of overweight is presented by Gabarró et al., which study the problem of how to solve red-red clusters after multiple insertions in a relaxed balanced red-black tree [8]. They introduce a new relaxed balancing algorithm for red-black trees, which they call *hyperred-black tree*. The only allowed updates in a hyperred-black tree are insertions. Hyper-red nodes accumulate degrees of redness in contrast to chromatic trees where multiple blackness is accumulated by overweighted nodes. In order to allow also deletions, recently Messeguer and Valles [25] proposed a combination of hyperred-black trees and chromatic trees, which they call *hyperchromatic tree*. They extend chromatic trees by allowing any number of color units in the nodes.

As the survey above reflects, many different relaxed balancing algorithms have been suggested. However, with regard to the comparison of relaxed balancing algorithm with other concurrent search tree algorithms and an experimental performance analysis, the B^{link}-tree is the only well studied relaxed balancing algorithm. There are analytical examinations and simulation experiments which show that the B^{link}-tree has a clear advantage over other concurrent but non-relaxed balanced B-tree algorithms [12, 13, 33]. So far the performance of concurrent relaxed balanced binary search trees has hardly been studied by experiments. Non of the experimental research known to us examine relaxed balanced binary trees in concurrent environments.

Bougé et al. perform some experiments in order to study the practical worst case and average convergence time of their relaxed balancing scheme for AVL-trees [3, 4]. They generate the sequence of all binary trees of 6, 8, or 10 nodes respectively. For each tree type they count how many steps are needed in average to balance the tree, where the rebalancing transformations are performed at randomly chosen nodes. Furthermore, they determine the convergence time of random trees of at most 500 nodes.

Analogously, Gabarró et al. study the practical behaviour of hyperred-black trees [8]. In order to compare hyperred-black trees experimentally with chromatic trees, they generate trees of size at most 100 and count the number of steps that are needed to obtain a red-black tree. At each step a node is picked up randomly and, if possible, an appropriate rebalancing transformation is carried out. The comparison of the results show that hyperred-black trees progress better than chromatic trees, although hyperred-black trees needs more rebalancing transformations. This is caused by the fact that the rate of failure to apply a rebalancing transformation at the randomly chosen node is much higher in chromatic trees than in hyperred-black trees.

Malmi presents some sequential implementation experiments applying his periodical rebalancing algorithm to the chromatic tree [23]. This algorithm is compared with an analogous periodical rebalancing algorithm for hight-valued trees [24]. The experimental results indicate that the algorithm for hight-valued trees is considerably faster than the algorithm for chromatic trees. In addition, Malmi compares experimentally the performance of the dictionary operations of chromatic trees and height-valued trees with the one of AVL-trees and some other non-relaxed balanced trees. In this (sequential) tests AVL-trees perform always very well. Furthermore, the efficiency of group-operations is compared with single operations, and it is shown that group updates provide a method of speeding up search and update time per key considerably [24].

This paper presents an experimental comparison of relaxed balanced binary search trees in a concurrent environment. We examine three relaxed balancing algorithms for red-black trees, the chromatic tree [6, 26, 27], the algorithm that is got by applying the general method of how to make strict balancing schemes relaxed to red-black trees [11], and the relaxed red-black tree [17]. The three algorithms are relatively similar and have the same analytical bounds on the number of needed rebalancing transformations. The chromatic tree and the relaxed red-black tree can both be seen as a tuned version of the algorithm got by the general relaxation method. However, it is unclear how much worse the performance of the basic relaxed balancing algorithm really is. Furthermore, from a theoretical point of view it is assumed that the relaxed red-black tree has an advantage over the chromatic tree, since its set of rebalancing transformations is smaller and the weight-balancing transformations lock fewer nodes at a time. But whether in practice the relaxed red-black tree really fits better is still unknown.

In order to clear this points, our goal was to compare the performance of the algorithm on large data in a concurrent environment. We developed the relief algorithms that are necessary to implement relaxed balancing algorithms in a concurrent environment—this includes the management of the background rebalancing processes and a suitable mechanism for concurrency control—, and then we implemented the algorithms using the simulation of a multi processor machine. By a series of experiments we compared the performances of the three algorithms in a highly dynamic application of large data. In order to warrant the use of relaxed balanced binary search trees in concurrent environments, we have also implemented the strictly balanced red-black tree [31] combined with the locking scheme of Ellis [7] and compared it with the relaxed balancing solutions.

The results of our experiments indicate that in a concurrent environment the relaxed balancing algorithms have a significant advantage over the strictly balanced red-black tree. With regard to the average response time of the dictionary operations the three relaxed balancing algorithms perform almost identically. Regarding the quality of the rebalancing we find that the chromatic tree has the best performance.

The remainder of this paper is organized as follows. In Section 2, first we give a short review of the red-black tree algorithms that we want to consider. Then, the individual rebalancing operations of the three relaxed balancing algorithms are compared in order to point out what are the main differences and what is broadly similar. In Section 3.1 we propose an algorithm how to maintain a problem queue from which the rebalancing processes get their work, and in Section 3.2 we extend the locking scheme by Nurmi and Soisalon-Soininen [27, 28] in such a way that it is possible to use a problem queue without running the risk of generating deadlock situations by the rebalancing processes. Finally, in Section 4 we describe the performed experiments and present the results.

2 Concurrent Red-Black Tree Algorithms

In this section we give a short review of the red-black tree algorithms that we want to compare.

The trees in this paper are leaf-oriented binary search trees, which are full binary trees (each node has either two or no children). The keys (chosen from a totally ordered universe) are stored in the leaves of the binary tree. The internal nodes contain only routers, which guide the search through the structure.

2.1 The Standard Red-Black Tree

First, we consider the strictly balanced red-black tree algorithm proposed by Sarnak and Tarjan [31]. Since the relaxed balanced algorithm described in Section 2.2, 2.3, and 2.4 are relaxed extensions of this algorithm, in the following we refer to it also as the *standard red-black tree* or *standard algorithm*.

The nodes of a standard red-black tree are coloured red or black, respectively. As the balance conditions it is required that

1. each search path from the root to a leaf consists of the same number of black nodes,
2. each red node (except for the root) has a black parent, and
3. all leaves are black.

In order to insert a new key x into a strictly balanced red-black tree, first its position among the leaves is located and the leaf is replaced by an internal red node with two black leaves. The two leaves now store the old key (where the search ended) and the new key x. If the parent of the new internal node p is red (as well as p itself) then the resulting tree is no longer a standard red-black tree. In order to correct this and to restore the balance conditions again, a rebalancing procedure is called for the new internal node p. If p's parent has a red sibling, then p's parent and its sibling is colored black and p's grandparent (which must be black) red, cf. Appendix A. This may produce a new violation of the red constraint. The same transformation is repeated, moving the violation up the tree, until it no longer applies. Finally, if there is still a violation an appropriate transformation is carried out that restores the balance conditions again using at most one rotation or double rotation, cf. Appendix A.

In order to delete a key from a strictly balanced red-black tree, first the leaf is located where the key is stored and then the leaf is removed together with its parent. If the removed parent was black, then the red-black tree structure is now violated. It can be restored easily by a color flip, if the remaining node is red. Otherwise, the removal leads to the call of a rebalancing procedure for the remaining leaf, cf. Appendix A . Analogously as in the insert procedure, the rebalancing operation bubbles up the violation of the black constraint in the tree by using only color changes and finally applies an appropriate transformation that restores the balance conditions again using at most two rotations or a rotation plus a double rotation, cf. Appendix A.

Rebalancing after an insertion or deletion can be done in $O(1)$ rotations and $O(\log n)$ color changes, if n is the size of the standard red-black tree. Furthermore, the number of color changes per update is $O(1)$ in the amortized case [31].

2.2 The Basic Relaxed Balancing Algorithm

In this section we sketch the ideas of the general method to make known rebalancing algorithms relaxed in an efficient way [11, 19] and consider the relaxed balancing algorithm that is got by applying the method to the standard red-black tree described in the previous Section 2.1.

In order to relax a strict balancing scheme, the real insert and delete operations (i.e. searching of a leaf and the insertion or removal of an item, respectively) are uncoupled from the rebalancing procedures. Instead of calling the rebalancing procedure immediately after an update operation only a rebalancing request is deposited at the corresponding node. The rebalancing can be delayed arbitrarily and interleaved freely with search and update operations. Furthermore, the actual removal of a leaf is also uncoupled from the delete operation, so that the deletion of a key in the tree leads to a *removal request* only. The actual removal is considered to be a part of the rebalancing task. In order to minimize the number of nodes that are considered by a rebalancing process at a time, the original rebalancing procedures are split into small restructuring steps, that can be carried out independently and also concurrently at different locations in the tree. The only requirement is that no two of them interfere at the same location. This can be achieved very easily by handling requests in top-down manner if they would meet in the same area (otherwise, in any arbitrary order).

If the tree contains rebalancing requests, then there is always a transformation that can be carried out, since the topmost request in the tree can be settled without any interference of the other ones.

This simple method applied to the standard red-black tree leads to the following algorithm [11], which we will call the *basic relaxed balancing algorithm*:

The nodes of a relaxed balanced red-black tree are either red or black. Red nodes may have up-in requests, black nodes may have up-out requests, and leaves may have removal requests. As the relaxed balance conditions it is required that

1. for each search path from the root to a leaf, the sum of the number of black nodes plus the number of up-out requests is the same,

2. each red node (except for the root) has either a black parent or an up-in request, and

3. all leaves are black.

In order to insert a new key x into a basic relaxed balanced red-black tree, its position among the leaves is located. If the leaf has a removal request, then the removal request is abandoned and the key is (re-)insert at that leaf. Otherwise, the leaf is replaced by an internal node with two leaves that store the old key (where the search ended) and the new key x. If the leaf has an up-out request, then the up-out request is removed, and the leaf is replaced by an internal black node with two leaves. Otherwise, the leaf is replaced by a red node with two leaves, and an up-in request is deposited at the new internal node, since the insertion may introduce a violation of the relaxed balance conditions, cf. Appendix B.

In order to delete a key x its position among the leaves is located. If the leaf has a red parent with an up-in request (as the result of a previous insertion), then the up-in request is abandoned and the leaf is removed together with its parent. Otherwise, a removal request is deposited at that leaf, cf. Appendix B.

The task of rebalancing a basic relaxed balanced red-black tree is to handle all up-in, up-out, and removal requests. This can be done by using essentially the same set of rebalancing transformations as in the strict case.

A removal request can be handled, if neither the leaf nor its parent has an up-out request as result of a previously settled removal request. (This assumption assures that two removal requests appended to both leaves of a black node are always handled correctly.) In order to resolve a removal request, the leaf is removed together with its parent. If the parent was black, then if the remaining sibling is red, it is colored black, and otherwise an up-out request is deposited, cf. Appendix B.

An up-in or up-out request can be handled by applying one single restructuring step of the rebalancing procedure in Red-Black Tree Insertion or Red-Black Tree Deletion. So, either the request is settled by a local structural change or it is shifted to the grandparent (parent) and handled at a later time, cf. Appendix B. These transformations can be applied concurrently and in arbitrary order as long as they do not interfere at the same location. If requests would meet in the same vicinity, it is avoided by handling them in top-down manner.

In order to tune the basic relaxed balanced red-black tree two additional transformations are defined in [11], that apply in situations that cannot occur in a strict balanced red-black tree. If two sibling nodes have up-out requests, then the requests can be handled in the following way. If the parent p of those nodes is red then p is colored black and the up-out requests from both children of p are removed. Otherwise, the up-out requests of p's children are replaced by an up-out request for p (recursive shift). The second additional transformation is a combination of the transformation that applies if the node with the up-out request has a red sibling with the first additional transformation, cf. Appendix B.

Since essentially the same set of rebalancing transformations as used in the strict case are also used for the relaxed case, the known bounds from the strict balancing scheme carry over to relaxed balancing [11].

2.3 Chromatic Tree

The nodes of a *chromatic tree* [5, 6, 26, 27] are colored red, black, or overweighted. The color of each node p is represented by an integer value $w(p)$, the *weight* of p. The weight of a red node is zero, the weight of a black node is one, and the weight of an overweighted node is greater than one. As relaxed balance conditions it is required that

(i) all leaves are not red and

(ii) the sum of weights on each search path from the root to a leaf is the same.

Insertion and deletion in a chromatic tree can be done in the following way.

In order to insert a key x, we search for x in the chromatic tree. If the search ends up unsuccessful, then we replace the leaf with weight w by an internal node with weight $w-1$ and two black leaves as its children, cf. Appendix C. The two leaves now store the old key (where the search ended) and the new key x.

In order to delete a key x from a chromatic tree, we locate the leaf where the key is stored. If the search ends up successful, then we remove the leaf together with its parent and increase the weight of the remaining sibling by the weight of the parent node, Appendix C.

Inserting a key may introduce two adjacent red nodes on the search path from the root to the inserted leaf. We say there is a *red-red conflict* at a node p, if p is red and has a red parent. Deleting a key may introduce an overweighted node. This is called an *overweight conflict*.

Since a chromatic tree that contains no red-red and no overweight conflicts clearly fulfills the balance conditions of the standard red-black tree, the task of rebalancing a chromatic tree is to remove all red-red and overweight conflicts from the tree. For this purpose Nurmi and Soisalon-Soininen define a set of rebalancing operations for chromatic trees [26, 27]. A more efficient set of transformations is defined by Boyar and Larsen [5, 6]. It includes four different types of transformations, cf. Appendix C.

1. The *blacking* transformation handles a red-red conflict by using only color changes. Thereby either the red-red conflict is resolved or it is shifted to the grandparent node.
2. The *red-balancing* transformations remove a red-red conflict by performing a rotation or a double rotation.
3. The *push* transformation handles an overweight conflict by using only color changes. Thereby either the overweight conflict is resolved or it is shifted to the parent node.
4. The *weight-decreasing* transformations remove an overweight conflict by performing a structural change (at most two rotations or a rotation plus a double rotation).

It is required that several red-red conflicts at consecutive nodes are handled in top-down manner. Despite of this exception, the rebalancing transformations can be applied in any arbitrary order.

If T is a chromatic tree of size n that fulfills the balance conditions of standard red-black trees and i insertions and d deletions are applied to T, then $O(i \cdot \log(n+i))$ blacking operations, $O(i)$ red-balancing transformations, $O(d \cdot \log(n+i))$ push transformation, and $O(d)$ weight-decreasing transformations are needed in order to restore the balance conditions of standard red-black trees again [6]. Furthermore, starting with an empty chromatic tree only an amortized constant amount of rebalancing is necessary after an update [5].

2.4 Relaxed Red-Black Tree

Analogously as the basic relaxed balancing algorithm and the chromatic tree, a *relaxed red-black tree* [17] is a relaxation of the standard red-black tree. The relaxed balance conditions, the operations insert and delete, and the transformations to handle a red-red conflict are exactly the same as of chromatic trees, cf. Section 2.3. The set of rebalancing transformations to handle an overweight conflict is reduced to a smaller collection of operations, which perform at most one rotation or double rotation each, cf. Appendix D. It includes three types of transformations:

1. The *weight-push* transformation handles an overweight conflict by using only color changes. Thereby either the overweight conflict is resolved, or it is shifted to the parent node.

2. The *weight-decreasing* transformations remove an overweight conflict by performing a structural change (at most one rotations or one double rotation).

3. The *weight-temp* transformation performs a rotation in order to prepare a weight-decreasing transformation but does not resolve an overweight conflict itself.

For relaxed red-black trees the number of structural changes after an update is also bounded by $O(1)$ and the number of color changes by $O(\log n)$, if n is the size of the tree. Furthermore, the rebalancing is amortized constant [17].

2.5 Comparison of the Algorithms

In the following we compare the rebalancing operations of the three relaxed balancing algorithms described in the previous sections in order to point out what are the main differences and what is broadly similar.

An up-out request is equivalent to one unit of overweight. In the chromatic tree or the relaxed red-black tree the weight of a node can take any positive integer value, whereas in the basic relaxed balancing algorithm the weights of nodes are bounded by two. This restriction leads to the need of removal requests. Therefore, in the basic relaxed balancing algorithm in general the actual removal of leaves are part of the rebalancing task, whereas in the case of chromatic trees and relaxed red-black trees the actual removal is part of the update operation delete.

An up-in request, which marks a red node with a red parent, is equivalent to a red-red conflict. The rebalancing transformations to handle a red-red conflict are identical for chromatic trees and relaxed red-black trees. These transformations can be seen as generalizations of the transformations to handle an up-in request, since they differ only in the allowed units of overweight, cf. Appendix E. All three algorithms have in common that several red-red conflicts at consecutive nodes have to be handled in top-down manner.

The bound on the overweights leads to an additional restriction of the order in which transformations can be applied in the basic relaxed balancing algorithm. For example, if the grandparent of a node with an up-in request has an up-out request, then the up-out request has to be handled first. If exactly the same situation occurs in a chromatic tree or relaxed red-black tree, then the red-red balancing transformation can be carried out immediately.

The rebalancing transformations to handle an overweight conflict also resemble one another. Let us first compare the transformations of the basic relaxed balancing algorithm and the chromatic tree, cf. Appendix B and C. Despite of the (up-out a+c)-transformation in the case of the basic relaxed balancing algorithm and (w4)-transformation in the case of the chromatic tree, the weight transformations correspond exactly to each other and differ only in the allowed units of overweight, cf. Appendix E. The (up-out a+c)-transformation and the (w4)-transformation match each other in the sense that (w4) applies in all situations where (up-out a+c) applies, and both perform a structural change consisting of two rotations in order to resolve one unit of overweight. So, also the weight transformations of chromatic trees can approximatively be seen as a generalization of the transformations of the basic relaxed balancing algorithm to handle an up-out request.

Let us now consider the set of weight transformations of the relaxed red-black tree. It consists of only five operations instead of eight. Four of them, the weight-push transformation and the weight-decreasing transformations, are identical with operations of the chromatic tree, cf. Appendix E. The remaining transformation weight-temp does not resolves an overweight conflict but prepares one of the weight-decreasing transformations using one rotation and some color flips. If we assume, that a weight-temp operations is carried out not independently but followed immediately by a weight-decreasing transformation, then such a combined transformation is again almost identical with the corresponding transformation of the chromatic tree. That is, because weight-temp differs from the operation (up-out a) only in the allowed units of overweight. In contrast to the chromatic tree, in the relaxed red-black tree a weight-temp transformation can also be carried out independently and interleaved freely with other operations. This has the advantage that fewer nodes at a time have to be considered by a rebalancing process. On the other hand it introduces a restriction in which order weight-balancing transformations can be carried out, since a weight-temp transformation cannot be applied if a particular child of the red sibling of an overweighted node is overweighted as well, cf. Figure 12 (c). As we will see in Section 4.2.1 this restriction strongly impaires the rebalancing.

The detailed comparison of the transformations depicts that the three relaxed balancing algorithms are very similar. Chromatic trees and relaxed red-black trees can both be seen as generalizations of the basic relaxed balancing algorithms, where it is allowed to accumulate several up-out requests at a node in order to tune the basic algorithm.

3 Relief Algorithms

In this section we consider the additional algorithms that are necessary to implement the relaxed balancing algorithms in a concurrent environment. This includes the management of the background rebalancing processes and a suitable mechanism for concurrency control.

3.1 Administration of the Rebalancing Requests

If update operations are performed in a relaxed balanced tree, then at several nodes in the tree the balance conditions of the strict case may be violated. In order to mark such locations in the tree, the update operations deposit *rebalancing requests* at the corresponding nodes. This helps the rebalancing processes to locate the imbalance situations and to determine which transformation can be applied in order to restore the balance conditions again.

In the case of relaxed balanced red-black trees the rebalancing requests are implemented by the color of the node. One unit of overweight denotes a request for a weight-balancing transformation. Two consecutive red nodes denote a request for a red-balancing transformation.

In our implementation of chromatic trees and relaxed red-black trees we have also used up-in requests (as defined in the basic relaxed balancing algorithm, cf. Section 2.2) in order to mark red-red conflicts in the tree. This has the advantage that a red-red cluster can be detected by considering only the parent of a red node or, the other way round, that never the parent of a red node without an up-in request must be checked in order to exclude a red-red conflict. An upin request can also be implemented by using the color field. For this, introduce a new color "up-in request" and adapt the color changes of the rebalancing transformations suitably.

In the literature there are three different suggestions in which way rebalancing processes locate rebalancing requests in the tree. One idea is to traverse the data structure randomly in order to search for rebalancing requests [26, 27]. In each step a small region is considered (incipient with the top-most node and then in top-down direction) and it is checked whether a rebalancing transformation applies.

A periodical rebalancing algorithm is presented by Malmi [23, 24]. During the insert and delete operations all nodes are marked on the search path from the root to the leaf where the update occurs. This makes the search for rebalancing requests more efficient, since the rebalancer can restrict its search to the marked subtrees.

The third suggestion is to use a problem queue [5, 6, 16]. Whenever an imbalance situation in the tree is created, then a pointer to the top-most node involved is placed in a queue of the rebalancing processes. This has the advantage that rebalancing processes can search for imbalance situations purposefully. But in contrast to the other solutions, it is necessary for each node to have a parent pointer in addition to the left and right child pointers.

In order to implement the relaxed balancing algorithms we decided to administrate the rebalancing requests by a problem queue. For this purpose, we developed the following algorithm how to maintain a problem queue, and we suitably extended the locking scheme proposed by Nurmi and Soisalon-Soininen [26], which we will present in Section 3.2.

In addition to the color field of a node each rebalancing request is implemented by a pointer to the node. In the following we call this pointer a *request pointer*. Whenever a rebalancing request is generated, the corresponding request pointer is appended to a problem queue. Therefore, every rebalancing request in the tree is represented by an item of a problem queue. If the problem queue is empty (or, respectively, all problem queues are empty, if several queues are used), then the tree is balanced. Now it may occur that a rebalancing request is resolved automatically by an update or a rebalancing transformation applied to another request. So, the problem queue may contain pointers to nodes that have no rebalancing requests any more. Furthermore, the problem queue may also contain pointers to nodes that are no longer in the tree, since it is allowed to remove nodes which (or the parent or sibling of which) have rebalancing requests. In order to avoid side effects, we demand that every node can be represented only by exactly one item of a problem queue. Each node contains an additional bit that is set, when a request pointer is appended to a problem queue, and that is unset again by the rebalancing process that removes the request pointer from the queue. The bit prevents that several request pointers to the same node are generated.

If a node should be removed from the tree the bit of which is set, then we know that a rebalancing process will consider this node at a later time. Therefore, in this case we mark the node after the removal explicitly as removed (for example by an invalid key), so that the rebalancing process notices that it is no longer part of the tree. Furthermore, if the implementation does not use a garbage collector but frees the allocated memory itself, then, of course, the node cannot be disposed by the removal operation. The freeing of the memory allocated for the node is delayed and becomes part of the rebalancing task.

User processes as well as rebalancing processes append request pointers at the tail of a problem queue, while pop operations at the head of the queue are carried out only by rebalancing processes. In order to assure a higher degree of concurrency it should be possible to access the head and the tail of the queue at the same time without interferences. For this reason, we have implemented the problem queue in the following way by using a single-linked linear list with two dummy elements. The pointers to the head and the tail of the queue are protected by exclusive locks. (Note, that the lock for the head pointer is not necessary if only one rebalancing process uses the problem queue.) In order to append a request pointer to the problem queue, first the lock of the tail pointer is acquired. Then, the request pointer is inserted into the dummy element, and a new dummy element is created at the end of the queue. After that, the tail pointer is updated and the lock is released. In order to perform a pop operation, first the lock of the head pointer is acquired (if necessary). Then it is tested whether the queue is empty by checking if the next pointer of the head dummy element is equal with the tail pointer. (This means, the value of the tail pointer is considered without using the corresponding lock. Therefore, in order to guarantee consistency, it must be assured that an append operation updates the tail pointer not until it has inserted the request pointer and created the new dummy element.) If the pointers are different, then the dummy element has at least one non-trivial successor item. The successor item is removed and its request pointer is returned after releasing the lock of the head pointer.

3.2 Concurrency Control

In this section, first we review the locking scheme presented by Nurmi and Soisalon-Soininen [27, 28], and then we present the extensions that are necessary to implement the rebalancing when using a problem queue.

The locking scheme by Nurmi and Soisalon-Soininen [26, 27] uses three different kinds of locks, *r-locks*, *w-locks*, and *x-locks*. Several processes can r-lock a node at the same time. Only one process can w-lock or x-lock a node. Furthermore, a node can be both w-locked by one process and r-locked by several other processes, but an x-locked node cannot be r-locked nor w-locked. Finally, a w-lock can be converted into an x-lock if there is no r-lock on the node, an x-lock can be converted into a w-lock, and a w-lock can be converted into an r-lock.

An r-lock on a node assures that the parts of the node that are read during a search operation (i.e. the key of the node and the pointers to its children) cannot be changed by an update or rebalancing operation. Update and rebalancing processes that want to exclude other update and rebalancing processes but no search processes use w-locks. This makes sense if, for example, a process changes only the colour of a node. Whenever a process changes the search structure of the tree then it uses x-locks in order to exclude all other processes from the critical nodes.

A process performing a search operation uses r-lock coupling from the root. (*Lock coupling* means that a process on its way from the root down to a leaf locks the child to be visited next before it releases the lock on the currently visited node.) At the leaf the process checks whether the key is found and reports success or failure. After that, the r-lock is released and the search terminates.

A process that performs an update operation uses w-lock coupling during the search phase. If the update operation is an insertion, then the lock of the parent of the leaf where the search ends is released. The w-lock of the leaf is changed to an x-lock and the leaf is changed to an internal node with two leaves as its children. Finally, the x-lock is released, the process reports success and the insertion terminates. If the update operation is a deletion, then the sibling of the leaf that should be removed is w-locked as well. Afterwards the w-locks are converted to x-locks in top-down order. Then the leaf is deleted, the contents of the sibling is copied to the parent node, and then the sibling is deleted. After adjusting the weights the only remaining lock is released, the process reports success, and the deletion terminates.

If this locking strategy for update processes should be used in the basic relaxed balancing algorithm, then it must slightly be modified, since the delete operation only deposits a removal request at the corresponding leaf and the actual removal is part of the rebalancing task. So, in a basic relaxed balanced red-black tree each update operation uses w-lock coupling in order to locate the leaf, releases the lock of the parent node, and if the search was successful, changes the w-lock of the leaf to an x-lock. In the case of a deletion, afterwards the leaf is marked with a removal request, the x-lock is released, the process reports success, and the deletion terminates. In the case of an insertion, the actual insertion proceeds as described above, if the leaf has no removal request. Otherwise, it is sufficient to overwrite the key of the leaf with the new value and then to release the x-lock.

A rebalancer uses w-locks while checking whether a transformation applies. Just before the transformation, it converts the w-locks of all nodes the contents of which will be changed to x-locks. Rotations are implemented by exchanging the contents of nodes, so that the parent of the top-most node involved can freely be accessed by other processes. If a rebalancing process decides that the chosen rebalancing transformation does not apply, it releases immediately all locks and continues searching for other imbalance situations in the tree.

Nurmi and Soisalon-Soininen suggest that rebalancing processes locate rebalancing requests by traversing the tree nondeterministically, so that rebalancing processes can always lock the nodes incipient with the top-most one and then its successors. Therefore, all processes always lock nodes in top-down direction. This guarantees that the locking scheme is dead-lock free [27, 28].

If we want to use a problem-queue in order to administrate the rebalancing requests, the situation is different. The red-black tree rebalancing transformations have the property that the parent or even the grandparent of a node with a rebalancing request must be considered in order to apply a transformation. Therefore, if a rebalancer follows the reference of a request pointer when searching for an imbalance situation in the tree, it cannot lock the top-most node involved first. This makes it necessary to extent the locking scheme in a suitable way in order to avoid dead-lock situations. In the following we present a possible strategy for rebalancing processes if a problem queue is used.

After poping a request pointer from the problem queue, a rebalancer w-locks the corresponding node. If the node has already been removed from the tree, then it is disposed. Otherwise, the rebalancer unsets the bit that indicates that a request pointer exists. If the node has no rebalancing request any more, then the w-lock is released, and the rebalancing operation terminates. Otherwise, the rebalancer tries to w-lock the parent node and, if necessary, afterwards the grandparent node. If the parent (or grandparent) is still w-locked or x-locked by another process, then

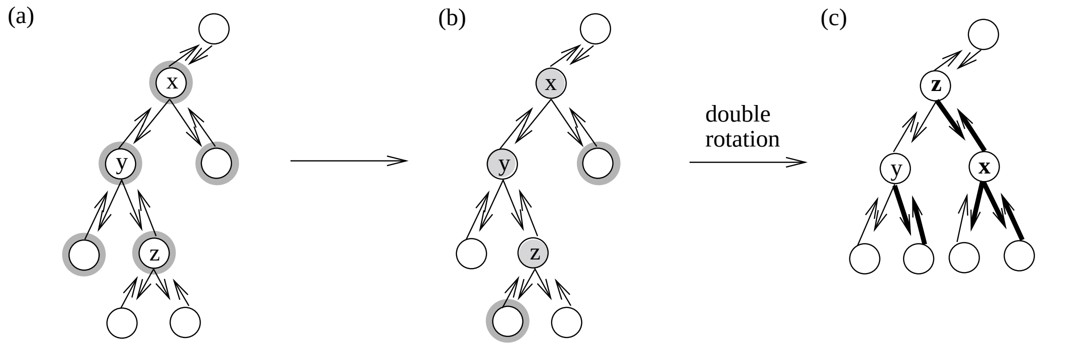


Figure 1: Example of implementing a structural change: (a) Check which of the transformations applies. (b) Prepare a double rotation. (c) Situation after performing the double rotation.
(A grey border around a node denotes a w-lock and a grey fill denotes an x-lock. Pointers and keys that have changed are printed bold.)

in order to avoid a dead-lock situation, the rebalancer sets the bit in the node again and releases all locks. Then, either it tries again to lock the necessary nodes, or it appends the request pointer to the problem queue and terminates the rebalancing operation unsuccessfully. If the lock(s) can be taken, then the rebalancer continues analogously as proposed by Nurmi and Soisalon-Soininen. It w-locks all other nodes that are necessary in top-down direction. If a transformation applies, it converts the w-locks of all the nodes the keys and child pointers of which are changed to x-locks in top-down direction and performs the transformation. If necessary, a request pointer to a new generated rebalancing request is appended to the problem queue. Afterwards all locks are released and the rebalancing operation terminates. If no transformation applies, for example if a red-red cluster is detected, then the rebalancer sets the bit in the node again, releases all locks, and appends the request pointer again to the problem queue.

As in the underlying locking scheme by Nurmi and Soisalon-Soininen all structural changes are implemented by exchanging the contents of nodes, in order to avoid that the parent of the top-most node involved must be x-locked. Since we use a problem queue for administrating the rebalancing requests, it is necessary for each node to have a parent pointer in addition. Therefore, more nodes must be locked during a rebalancing transformation than in the original locking scheme, in order to update also the parent pointers of child nodes, see Figure 1 for an example, but it is sufficient to w-lock these nodes, because only rebalancing processes use the parent pointers.

Since structural changes are implemented by exchanging the contents of nodes, in particular the address of the root node is never changed by a rotation. The only situation where the pointer to the root may be replaced is, if a key is inserted into an empty tree or the last element of the tree is deleted. This can be avoided by using as empty tree an one-element-tree with a removal request at its only leaf instead of a nil pointer. So, it is not necessary to protect the root pointer by a lock.

As the underlying locking scheme by Nurmi and Soisalon-Soininen our extension is also dead-lock free. All locks that may block the process are taken in top-down direction. Only rebalancing processes lock nodes in bottom-up direction. Since they do not wait for a lock of an ancestor node, if it cannot be taken, but release all other locks they hold immediately, dead-lock situations are avoided.

4 The Experiments

In order to compare the relaxed balancing algorithms for red-black trees, we have used a simulation of a multi-processor environment, *psim*, that was developed in co-operation with Kenneth Oksanen from the Helsinki University of Technology. In the following we give a short description of the simulator *psim*, and then we describe the experiments and present the results.

4.1 The Interpreter psim

The interpreter *psim* serves to simulate algorithms on parallel processors. The algorithms are formulated by using an own high-level programming language, that can be interpreted, executed, and finally statistically analyzed by *psim*.

psim simulates an abstract model of a multiprocessor machine, a CRCW-MIMD machine with shared constant access time memory [1]. The abstract *psim*-machine can be described as follows.

The *psim*-machine has an arbitrary number of (serial) processors, that share a global memory. All processors can read or write memory locations of the global memory at the same time. Each processor has local registers, where operands of arithmetic operations, interim results, etc. are stored. In order to read a value from the global memory into a local register or, respectively, to write a value from a register into the shared memory each processor needs one unit of *psim-time*. The execution of a single instruction (for instance an arithmetic operation, a compare operation, a logical operation, etc.) takes also one unit of psim-time each.

The access of global data can be synchronized by using semaphores or locks, where *psim* offers r-locks, w-locks, and x-locks. Several processes waiting for a lock are scheduled by a FIFO-queue. The time that is needed to perform a semaphore operation or a lock operation is constant each and can be chosen by the user. The overhead of lock contention is not modeled.

4.2 Experiments

Our aim was to study the behaviour of the different red-black tree algorithms in highly dynamic applications of large data. For this we implemented the algorithms in the *psim* programming language and performed concurrently 1000000 dictionary operations (insertions, deletion, and searches) on an initially balanced standard red-black tree which stored about 1000000 keys. As key space we choose the interval $[0, 2000000]$. The initial tree was built up by inserting 1000000 randomly chosen keys into an empty tree. During the experiments we varied the number of processes, the time that is needed to perform a lock operation, and the types of lock-modi that are available.

In each single experiment either 1, 3, 7, 15, 31, or 63 user processes apply concurrently altogether 1000000 random dictionary operations (insertions, deletions, and searches) to the tree. The probabilities that a dictionary operation is an insert, delete, or search operation are $p_{insert} = 0.5$, $p_{delete} = 0.2$, and $p_{search} = 0.3$ (analogously as in [13]). The rebalancing is done by a single rebalancing process that work concurrently with the user processes and gets its work from a problem queue, cf. Section 3.1. The offered lock modes are either all possible modes r, w, and x, or only r and x, or in the third variant only x. The costs of an arbitrary lock operation[1] are constant: either 1, 10, 20, 30, 40, 50, or 60 units of psim-time.

The choice of the lock costs between 1 and 60 is motivated by measurements of lock costs that we performed on several types of Sun and SGI machines. For this, we measured the time to execute a loop that acquires a lock, then waits some constant time, releases the lock, and again delays some time. (This test program is similar to that used by Anderson [2] or Lim and Agarwal [22].) In order to be able to transfer the measured time to the *psim*-model, we executed analogous loops for the arithmetic operations and converted the results to *psim*-time. Given in *psim*-time the measured lock costs were about 20 to 30 units of time.

4.2.1 Comparison of the Algorithms

In the first set of experiments the offered lock modes are r-locks, w-locks, and x-locks. We varied two parameters: the number of user processes and the lock costs.

The order in which rebalancing requests are handled depend on the order in which the user processes append them to the problem queue. If the rebalancing process creates a new rebalancing

[1] excluding the waiting time, if it is a lock acquire operation and the lock is already taken by another process

(a)

(b)

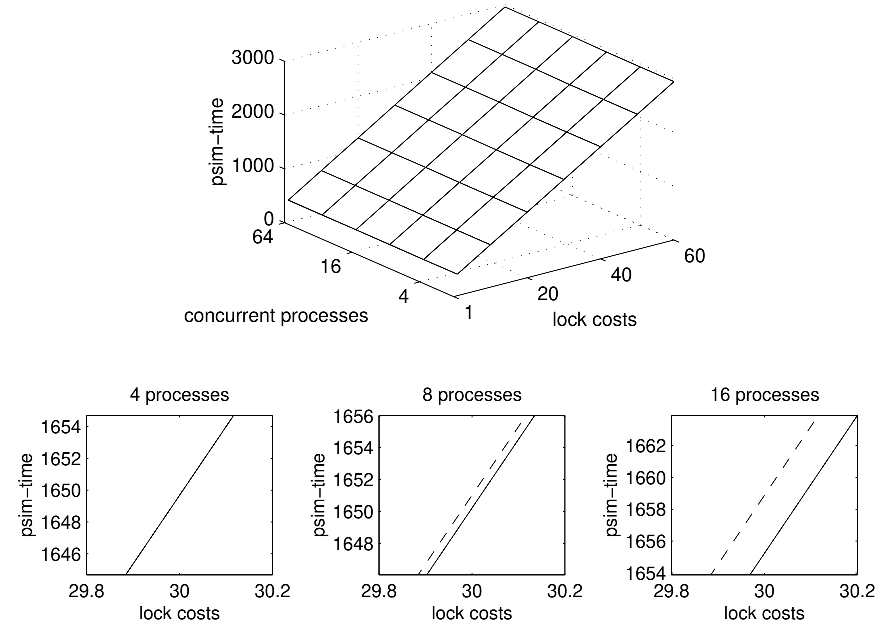


Figure 2: Average time needed to perform a *search operation.* (a) The whole diagram (b) Selected zooms. *Legend:* — — basic relaxed balancing algorithm, —— chromatic tree, — · — relaxed red-black tree

request, then it appends it to the tail of the problem queue as well and continues to handle another request from the head of the queue.

Figure 2–17 show the experimental results. The average time needed to perform a dictionary operation increases proportionally to the lock costs, where the amount of rise grows with the number of concurrent processes, cf. Figure 2–4.

The average time needed to perform a search operation is almost linear in the exponentially increasing number of concurrent processes, cf. Figure 2(a). The average time needed to perform an insert or delete operation grows strongly with the number of concurrent processes, cf. Figure 3 and Figure 4. Regarding the dictionary operations, the performances of the chromatic tree and the relaxed red-black are almost identical. In the case of the search and the insert operation, the performance of the basic relaxed balancing algorihm is only slightly lower. In the case of the delete operation, the basic relaxed balancing algorithms shows the tendency to need slightly less time than the other both algorithms, if the lock costs are small, and slightly more time, if the lock

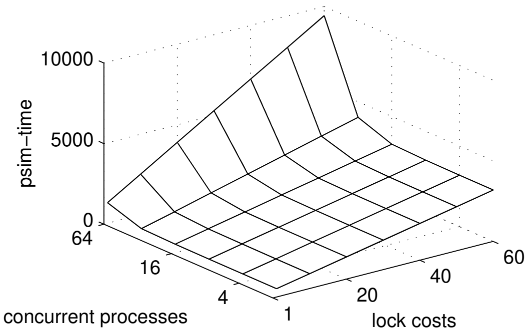
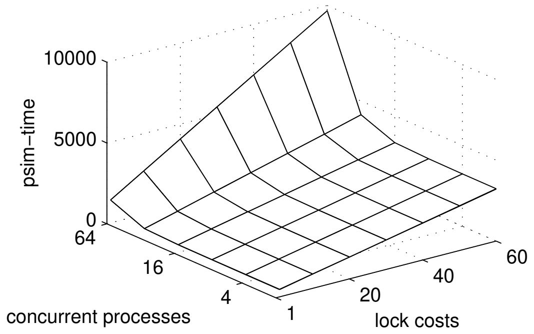


Figure 3: Average time needed to perform an *insert operation.*

(a)



(b)

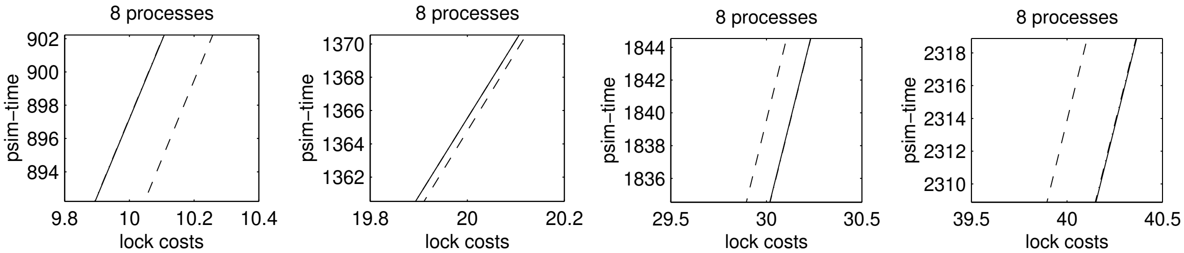


Figure 4: Average time needed to perform a *delete operation*: (a) The whole diagram (b) Selected zooms. *Legend:* – – basic relaxed balancing algorithm, —— chromatic tree, – · – relaxed red-black tree

costs are high. However, compared with the total time needed to perform one of the dictionary operations, in all three cases the amount is only insignificant.

For all three algorithms the average depth of a leaf is near the minimum depth of $\log_2 n$ a leaf can have: 21.7 in the case of two, four, and eight concurrent processes, and then it increases to 21.8 . This increase is caused by the proportion of one rebalancing process to a large number of user processes.

Figure 5 shows the comparison of the dictionary operations. The search is the cheapest operation. It is least sensitive to a growing number of concurrent processes. This is caused by the use of r-locks. The average time needed to wait for an r-lock is zero for all choices of lock costs and concurrent processes. In contrast to the search, the average time needed to perform an insert or delete operation rises strongly if the number of processes increases. This is caused by the use of w-locks during the search phase. The average costs for an update operation increase proportionally to the average time waiting for a w-lock, cf. Figure 6(a). Figure 5(b) depicts the average time needed to perform the actual insertion or deletion, if the leaf has already been found. It depends on the lock costs, which of the both update operations needs less time in average. In general, the actual deletion can be done faster, but if the lock cost are high, then it takes longer to perform an actual deletion than an actual insertion.

The total time needed in average to perform 1000000 dictionary operations is depicted in Figure 7. The results are very similar for all three algorithms. Almost always the chromatic tree terminates slightly earlier than the relaxed red-black tree. The basic relaxed balancing algorithm is sometimes a little faster and sometimes a little slower than the other both algorithms. However, compared with the total time needed to perform 1000000 dictionary operations, the differences between the three algorithms are insignificant.

As displayed in Figure 7(a) the total time needed in average to perform 1000000 dictionary operations decreases strongly from one user process to seven user processes, but then ends in a relative flat curve. This is directly connected with the utilization of the user processes, cf. Figure 8. In the case of lower or equal than 7 concurrent user processes the utilization of the user processes

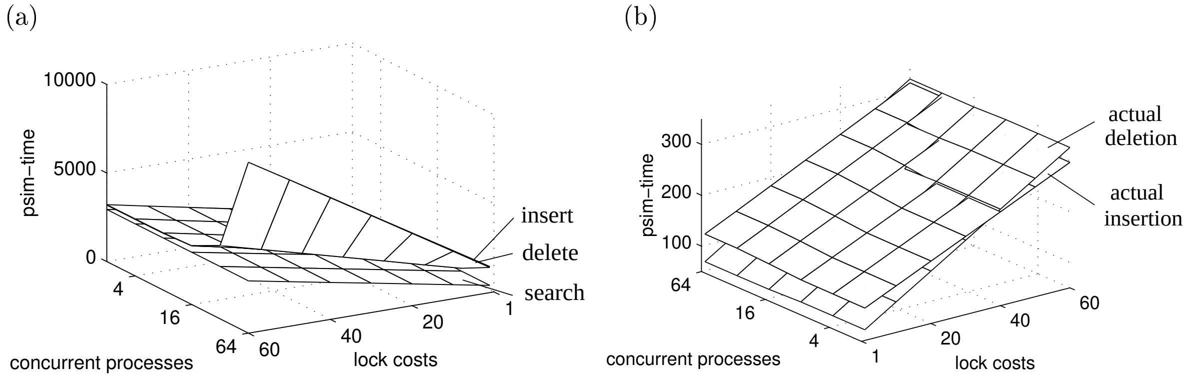


Figure 5: Comparison of the dictionary operations *insert*, *delete*, and *search* (exemplary shown by the relaxed red-black tree).

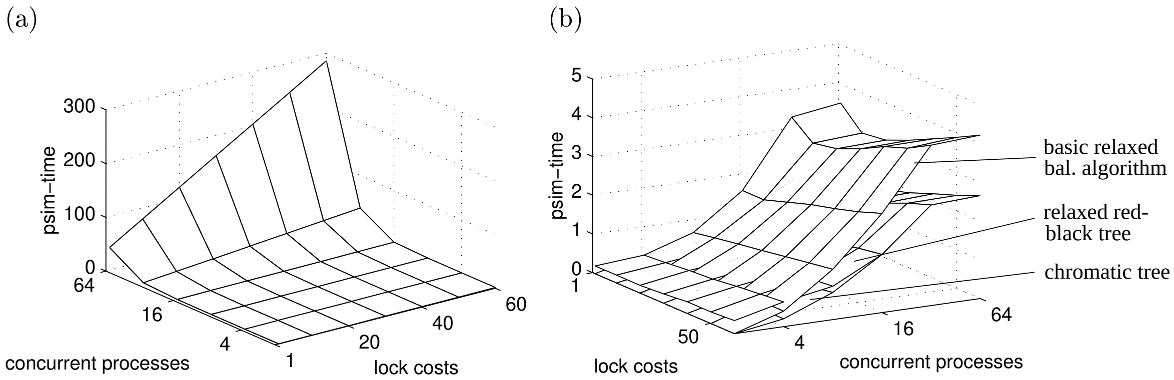


Figure 6: Average time needed to wait for a lock: (a) *w-lock*, (b) *x-lock*.

(a)

(b)

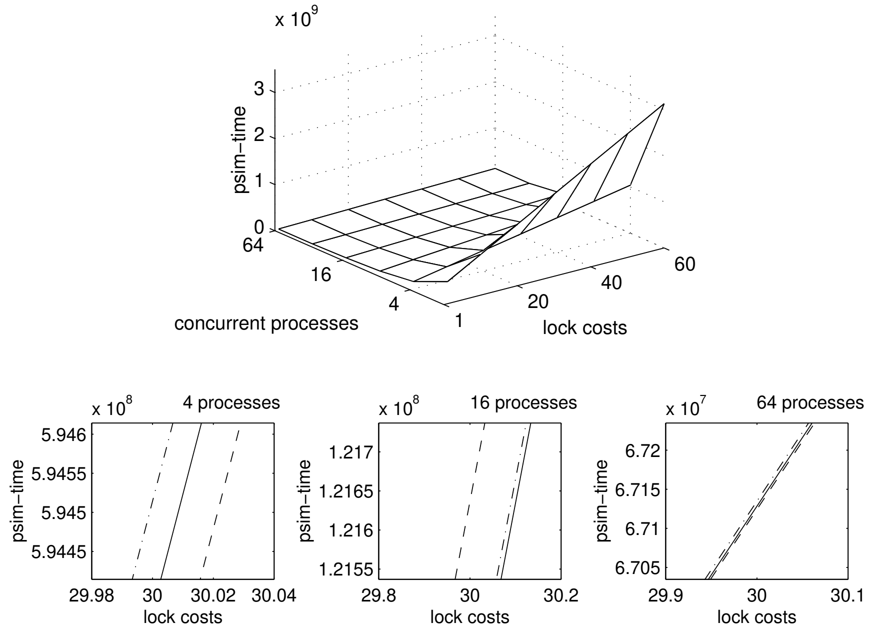


Figure 7: *Total time* needed in average to perform 1 000 000 dictionary operations: (a) The whole diagram (b) Selected zooms. *Legend:* – – basic relaxed balancing algorithm, —— chromatic tree, – · – relaxed red-black tree

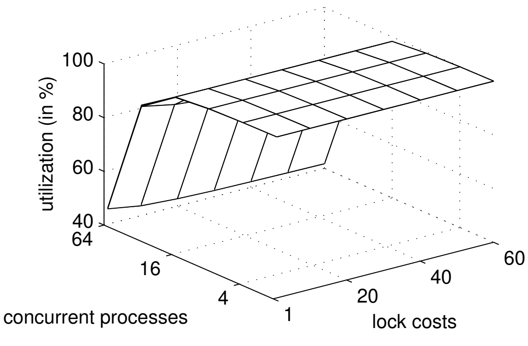


Figure 8: *Utilization* of the user processes.

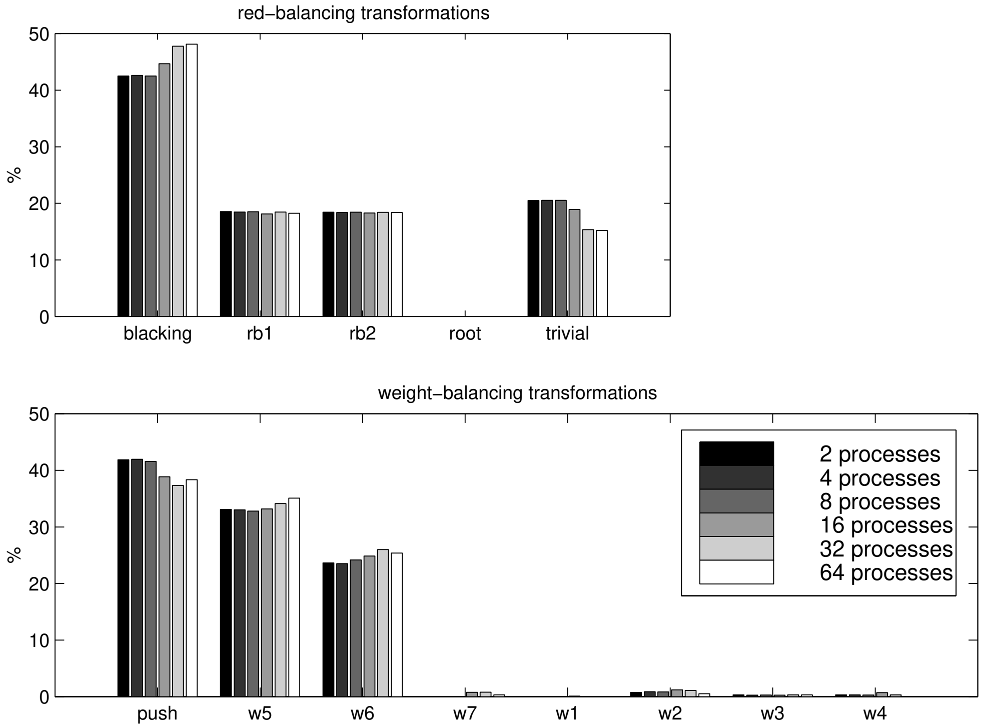


Figure 9: Proportion of the rebalancing transformations (exemplary shown by the chromatic tree).

is over 99%. From 15 to 31 user processes it decreases slowly to a value between 89% and 84% depending on the lock costs. In the case of 63 user processes the utilization is only between 46%, if lock costs = 1, and 42%, if lock costs = 60.

In contrast to the user processes, the utilization of the single rebalancing processes hardly changes, if the number of user processes or the lock costs are varied: it is always about 99%.

Figure 9 shows the proportion of the performed red-balancing transformations and, respectively, the proportion of the performed weight-balancing transformations. "trivial" denotes the trivial red-balancing transformation (upin a), that applies if a red node has an upin-request although its parent is black. It is noticeable that for all choices of the number of concurrent processes the occurence of one of the larger weight-balancing transformations is only very seldom.

As shown in Figure 10 (a), in the case of two concurrent processes there is almost no difference between the three relaxed balancing algorithms[2]. Furthermore, the rate of how often one of the larger transformations (w1)—(w4) occur is altogether only 1.36% of all weight-balancing transformations. This corresponds exactly to the occurrence of the weight-temp operation in the case of the relaxed red-black tree. With an increasing number of concurrent processes, some differences between the algorithms become clearly, mainly in the proportion of the smaller weight-balancing transformations, cf. Figure 10 (b).

Let us now consider the performance of the rebalancing operations. The average time needed to perform a rebalancing operation increases proportionally to the lock costs and is almost linear in the exponentially increasing number of concurrent processes, cf. Figure 11–15.

In contrast to the dictionary operations, the rebalancing operations also w-lock nodes in direction from the leaves to the root. In order to avoid deadlock situations, we have used the strategy that a rebalancer only tries to w-lock the parent of a node without blocking and releases immediately all locks that it holds, if the parent is still w-locked or x-locked by another process, cf.

[2]In Figure 10 the notation of the rebalancing transformations is chosen from the chromatic tree. See Appendix E for the corresponding names of the other both algorithms.

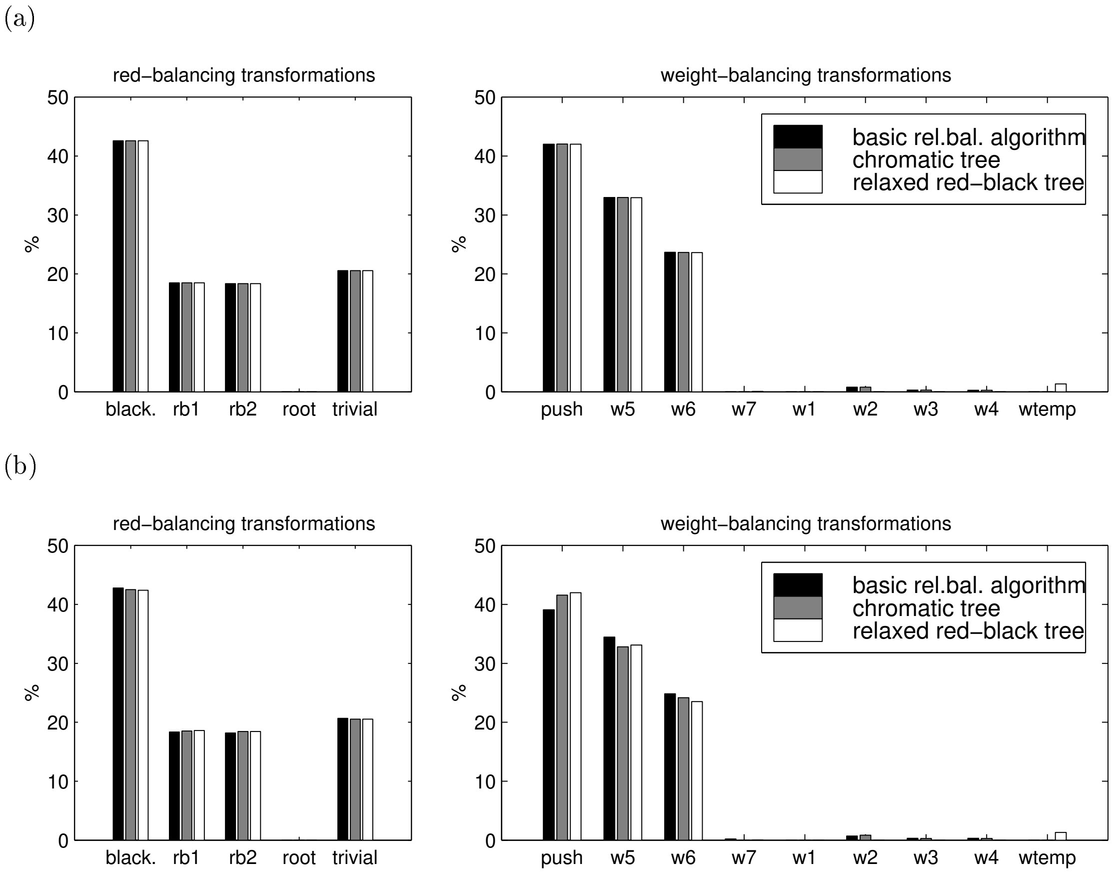


Figure 10: Proportion of the rebalancing transformations: Comparison of the relaxed balancing algorithms in the case of (a) 2 concurrent processes and (b) 8 concurrent processes.

(a)

(b)

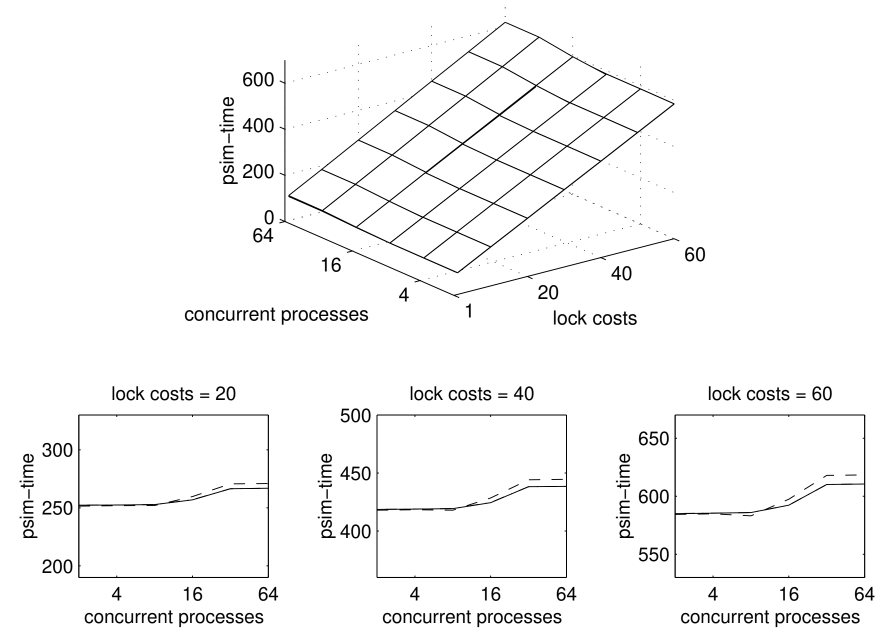


Figure 11: Average time needed to *handle a red-red conflict* (inclusive failures).
Legend: — — basic relaxed balancing algorithm, —— chromatic tree, — · — relaxed red-black tree

Section 3.2. Our experiments display that this strategy seems to be very suitable—at least in large trees. In 99.99% of all cases the w-lock could be taken successfully.

The average time needed to handle a red-red conflict is shown in Figure 11. Again the values of the chromatic tree and the relaxed red-black tree are almost identically. The average running time of a red-red balancing transformation of the basic relaxed balancing algorithm is slightly lower than the ones of the chromatic tree and the relaxed red-black tree, if 1, 3, or 7 user processes work concurrently (for all lock costs). In the case of 15, 31, and 63 user processes the value is always greater than the ones of the other both algorithms. This behaviour is illustrated in Figure 11(b).

Furthermore, we have counted how often a red-red balancing failed because of a red cluster. In the case of the chromatic tree and the relaxed red-black tree the rate was always under 0.001%. In the case of the basic relaxed balancing algorithm the rate was also under 0.001% for few processes and slightly higher for many processes, but in average always under 1%.

The handling of an overweight conflict is where the basic relaxed algorithm, the chromatic tree, and the relaxed red-black tree differ at most. Figure 13 depicts how often the handling of an overweight conflict failed because of an interfering conflict in the immediate vicinity (all possible cases are listed in Figure 12). In the case of the chromatic tree a failure of an overweight handling never occurred. In the case of the basic relaxed balancing algorithm the number of failures to handle an overweight conflict was almost always very low and at most 2.7% of all overweight handlings. The relaxed red-black tree differs crucially from the other both algorithms. Here the rate of failure to handle an overweight conflict was up to 99% of all overweight handlings. This occurrence depends on the weight-temp operation. Using a simple problem queue it seems to be non trivial to find an appropriate order in which overweight conflicts can be handled correctly. In all cases where the tree was relatively well balanced, most of the time the rebalancer tried to prepare a weight-temp operation, failed because of an overweighted child of the red sibling, and appended the request again to the problem queue. The following example is typical: In this experiment we used two processes, one user process and one rebalancing process, and chose the lock costs = 40 units of psim-time. During the time, when the user process performed 1000000 dictionary operations, the rebalancing process handled almost all red-red and overweight conflicts

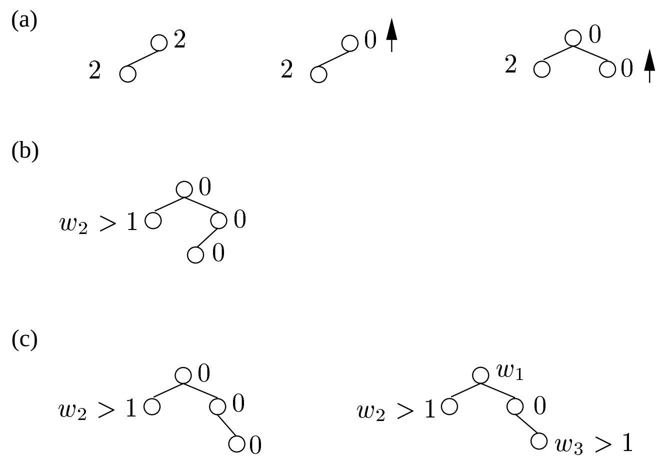


Figure 12: Possible cases when the handling of an overweight conflict fails (symmetric cases are omitted): (a) basic relaxed balancing algorithm, (b) chromatic tree, (c) relaxed red-black tree

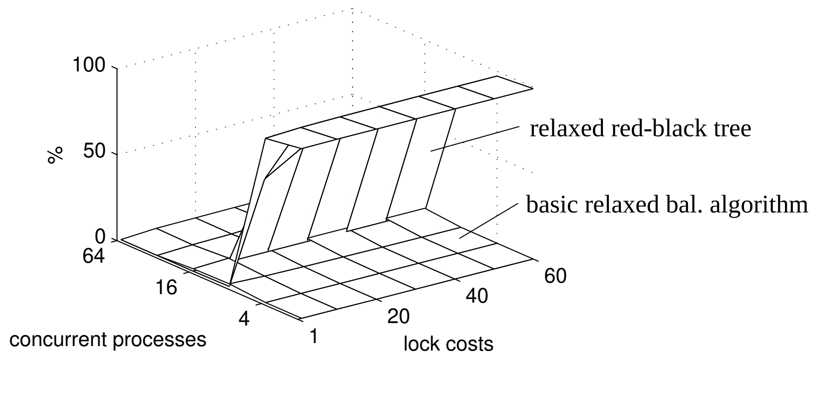
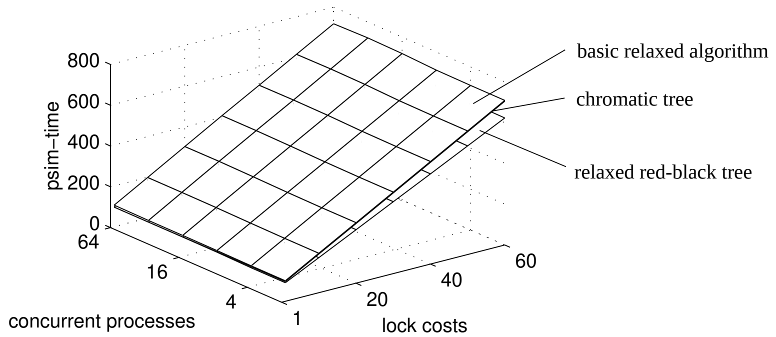


Figure 13: *Rate of failure* to handle an overweight conflict.
Legend: – – basic relaxed balancing algorithm, —— chromatic tree, – · – relaxed red-black tree

(a)



(b)

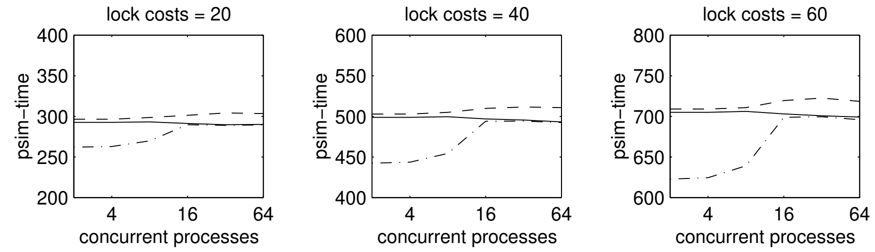


Figure 14: Average time needed to *handle an overweight conflict* (inclusive failures).
Legend: — — basic relaxed balancing algorithm, ——— chromatic tree, — · — relaxed red-black tree

in the tree. The number of successfully handled overweight conflicts was 17791 in total, where the number of successfully performed weight-temp operations was only 242. In contrast, 1988260 times the rebalancing process failed when trying to handle an overweight conflict at a node with a red sibling.

Figure 14 shows the average time needed to handle an overweight conflict. The basic relaxed balancing algorithm needs always slightly more time to perform a weight-balancing transformation than the chromatic tree. Note, that the curve of the relaxed red-black tree is not very meaningful in the case of 2, 4, and 8 concurrent processes because of the very high rate of failure. In the case of 16, 32, and 64 concurrent processes, where the rate of failure is only minimal, the chromatic tree and the relaxed red-black tree perform broadly similar. This is caused by the fact, that the cases where the handlings of an overweight conflict differ in both algorithms occur only very seldom, cf. Figure 9 and 10.

In order to complete the comparison of the rebalancing operations, Figure 15 shows the average time needed to handle an overweight conflict versa the average time needed to handle a removal request. As it was to be expected, the diagram of the average running time of the transformation to handle a removal request looks very similar as the ones of the other rebalancing transformations, and the handling of a removal request is much faster than the handling of an overweight conflict.

Finally, we consider the capacity of the rebalancing process. Figure 16 shows how much of the time, the rebalancing process spent idling because of an empty problem queue. For 8 or more concurrent processes, the rebalancing processes are always working to capacity. For less or equal than 8 processes, always the chromatic tree needs the least rebalancing work, so that the rebalancing process has the highest rate of idle-time. The case of two concurrent processes depicts the differences between the algorithms most clearly. With an growing number of concurrent processes the idle-time increases slightly from 82.66% to 88.46% in the case of the chromatic tree, from 75.47% to 85.11% in the case of the basic relaxed balancing algorithm, and from only 38.62% to 41.47% in the case of the relaxed red-black tree. This relative low idle-time of the rebalancing

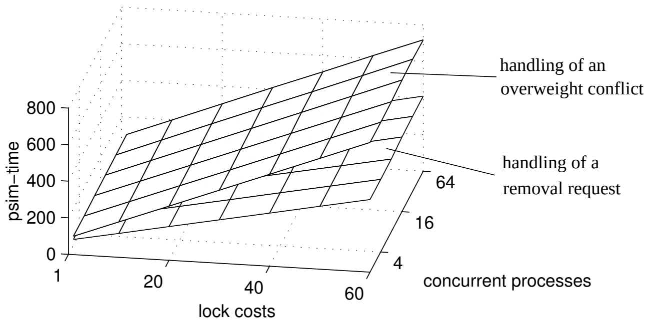


Figure 15: Basic relaxed balancing algorithm: rebalancing after a deletion (average time).

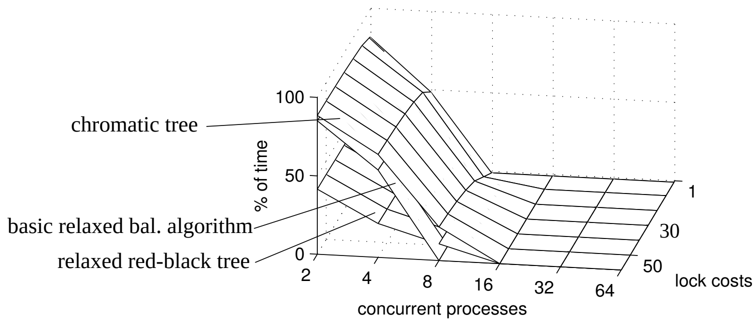


Figure 16: Idle-time of the rebalancing process.

process in a relaxed red-black tree reflects again the high rate of failure to handle an overweight conflict.

Another kind of measure of how well the tree is balanced by the single rebalancing process is the size of the problem queue after performing 1000000 dictionary operations and halt. Figure 17 shows the number of unsettled rebalancing requests. If eight or more concurrent processes are used, then the basic relaxed balancing algorithm remains always the most rebalancing requests. Despite of the slightly higher average time needed to handle a rebalancing request, this is caused by the additional work that the rebalancing process must do in order to handle also removal requests. The chromatic tree and the relaxed red-black tree leave almost the same number of requests, where the number by the relaxed red-black tree is always slightly higher than the one by the chromatic tree.

Figure 16 and Figure 17 depict that if only one single rebalancing process works in the background, then the use of seven concurrent user processes seems to be a suitable choice: In the case of eight concurrent processes the tree is well balanced, while the rebalancer works to capacity and the utilization of the user processes is over 99%, cf. Figure 8.

Altogether, the experimental results indicate that with regard to the average response time of the dictionary operations the performance of the three relaxed balancing algorithms is comparably good. Regarding the quality of the rebalancing, the chromatic tree has always the best performance. The rebalancing of a relaxed red-black tree seems to be very complicated because of the high rate of failure to perform a weight-temp operation, and the hoped speed up of the weight-balancing is only insignificant. The performance of the basic relaxed balancing algorithm is lower than the performance of the chromatic tree but still satisfactory. This gives grounds for the assumption that the general method of how to make strict balancing schemes relaxed generates satisfactory relaxed balancing algorithms, if it is applied to other classes of search trees as well.

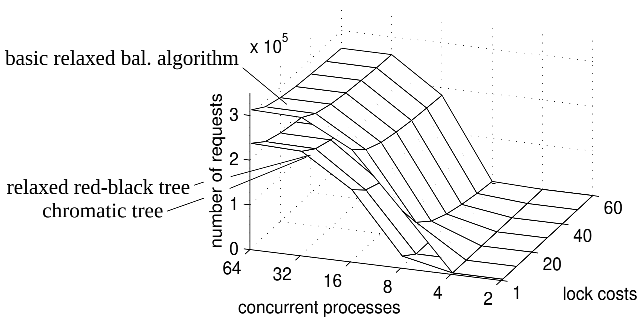


Figure 17: Number of unsettled rebalancing requests.

4.2.2 Relaxed Balancing makes sense

In the following we confirm by experiments that relaxed balancing has a significant advantage over strict balancing in a concurrent environment. For this, we implemented the standard red-black tree (cf. Section 2.1) using the locking scheme of Ellis [7] and performed an analogous series of experiments as for the relaxed balancing algorithms. Since in the standard red-black tree the rebalancing is done by the user processes, here we used 2, 4, 8, 16, 32, and 64 user processes instead of 1, 3, 7, 15, 31, and 63 user processes plus one rebalancing process. Figure 18–20 show the experimental results, where the standard red-black tree is compared with the chromatic tree.

The search operation is the only dictionary operation, where the standard red-black tree supplies almost the same results as the chromatic tree, cf. Figure 18 (a). In the case of the insert and the delete operation the chromatic tree has a significant advantage over the standard red-black tree, cf. Figure 18 (b) and (c).

The relative high average time that a standard red-black tree needs to perform one of the dictionary operation has also a negative effect on the total time needed to perform 1000000 dictionary operations, cf. Figure 19. Only in the case of two concurrent processes the standard red-black tree terminates earlier than the chromatic tree.

The utilization of the user processes reflects, why the standard red-black tree comes off so badly, cf. Figure 20. Since during an update operation the whole search path from the root to a leaf must be w-locked, the root of the tree becomes a bottleneck. This has the effect, that with an increasing number of concurrent processes the utilization of the processes decreases in a steep curve towards 4% only.

Altogether, the experimental results confirm the assumption that it makes sense to use relaxed balancing instead of strict balancing in a concurrent environment.

4.2.3 Variation of the supported lock modes

In this section we study in which way the behaviour of the relaxed balancing algorithms change, if less kinds of lock modes are supported. For this, we have tested the two variants that either only r- and x-locks are available but no w-locks or that x is the only supported lock mode. This means, for the first variant we have replaced all w-locks of the locking scheme described in Section 3.2 by x-locks and left out the conversion of w-locks to x-locks. The second variant differs from the first one in that also all r-locks used in the search operation are replaced by x-locks.

In most cases the results of these experiments look very similar as the results of the first set of experiments. Especially, the comparison of the three relaxed balancing algorithms leads to analogous results. Therefore, in the following we restrict ourself to present only the main differences to the results of the first set of experiments.

Figure 21 show the average time needed to perform one of the dictionary operations. It is remarkable that for less and equal than 16 concurrent processes the algorithms seem to be relatively resistant to a diminution of the available lock modes. In the case of the search operation

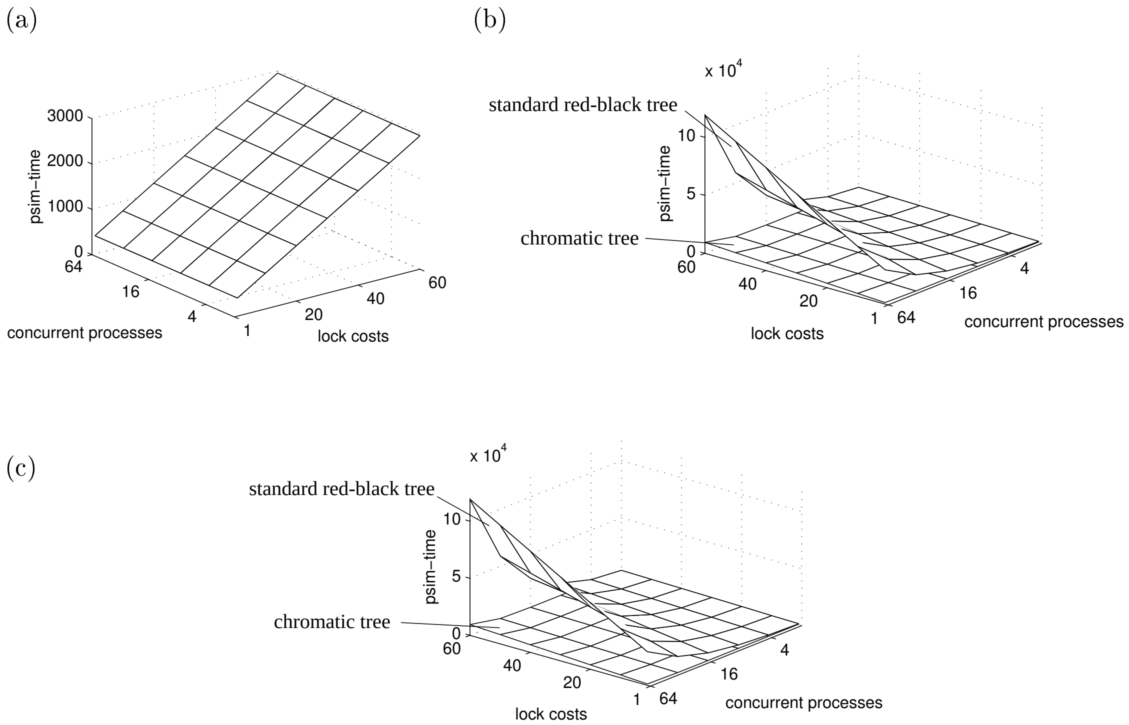


Figure 18: Comparison of the standard red-black tree and the chromatic tree. Average time needed to perform one of the dictionary operations: (a) search, (b) insertion, and (c) deletion.

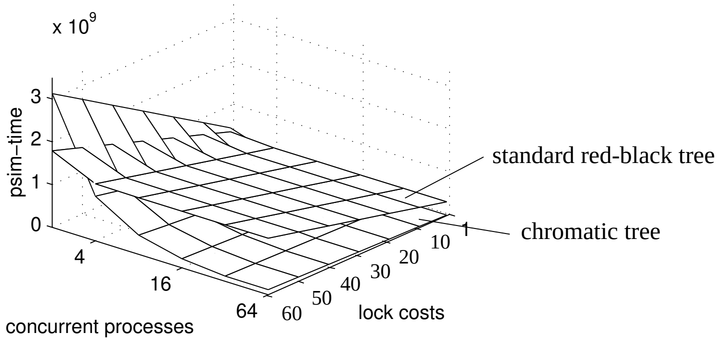


Figure 19: *Total time* needed in average to perform 1 000 000 dictionary operations.

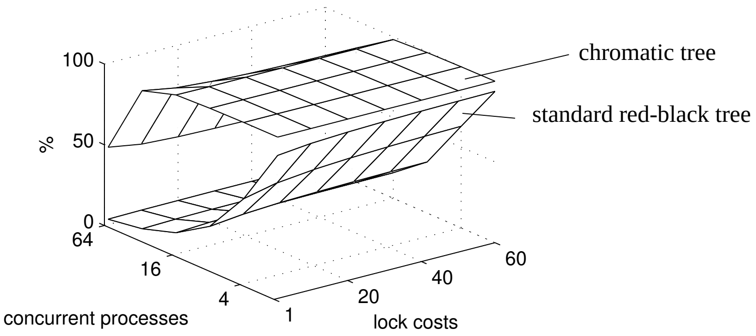


Figure 20: Utilization of the user processes.

the average time is only slightly higher, if no w-locks or x-locks are available. In the case of the update operations the average time is even slightly lower, because the conversion of w-locks to x-locks can be economized. However, if more than 16 concurrent processes are used, then there are plain differences. The version that all three lock modes r, w, and x are supported has always a clear advantage over the other ones.

The search operation changes mostly under a diminution of the supported lock modes, cf. Figure 21 (a). If only x-locks are available, then the curve showing the average time needed to perform a search operation looks very similar to the ones of the insert and delete operation, cf. Figure 22.

Figure 23 shows the total time needed in average to perform 1 000 000 dictionary operations. Analogously as for the update operations the curves look very similar in the case of lower or equal than 16 concurrent processes, cf. Figure 23 (a). The curves of the versions where only r- and x-locks or, respectively, only x-locks are available are even sightly lower than the curve of the full version. If more than 16 processes are used, than there is again a clear advantage of the full version over the other both versions, cf. Figure 23 (b).

The utilization of the user processes is depicted in Figure 24. The diagram confirms that for less or equal than 16 concurrent processes the processes do not disturb one another so much, that a diminution of the available lock modes would have a strong influence on the average time needed to perform one of the dictionary operations.

In contrast to the dictionary operations, the handling of a balance conflict is done always much faster in average, if no w-locks are available, cf. Figure 25. This is probably caused by the fact that the conversion of w-locks to x-locks is economized.

The lower costs to perform a rebalancing transformation results in that the rebalancing process settles more rebalancing request during the same time. Furthermore, the rebalancing process has more time to do its work, if less lock modes are available, since it takes longer to perform a dictionary operation. Therefore, in the case of greater or equal than 16 concurrent processes the size of the problem queue after performing 1 000 000 is clearly lower, if less kinds of lock modes are supported, cf. Figure 26.

4.2.4 Alternative order to handle red-red conflicts

This set of experiments differs from the first one only in the order in which red-red conflicts are handled. In the first set of experiments we used the strategy to append each newly created up-in request to the end of the problem queue after performing a blacking transformation (also called red-push transformation) and then to handle an arbitrary other request from the queue. The strategy used in this set of experiments is more similar to the sequential case: after performing a blacking transformation the rebalancing process continues to handle the newly created up-in request, if possible.

The results of both sets of experiments are broadly similar, except for the average time needed

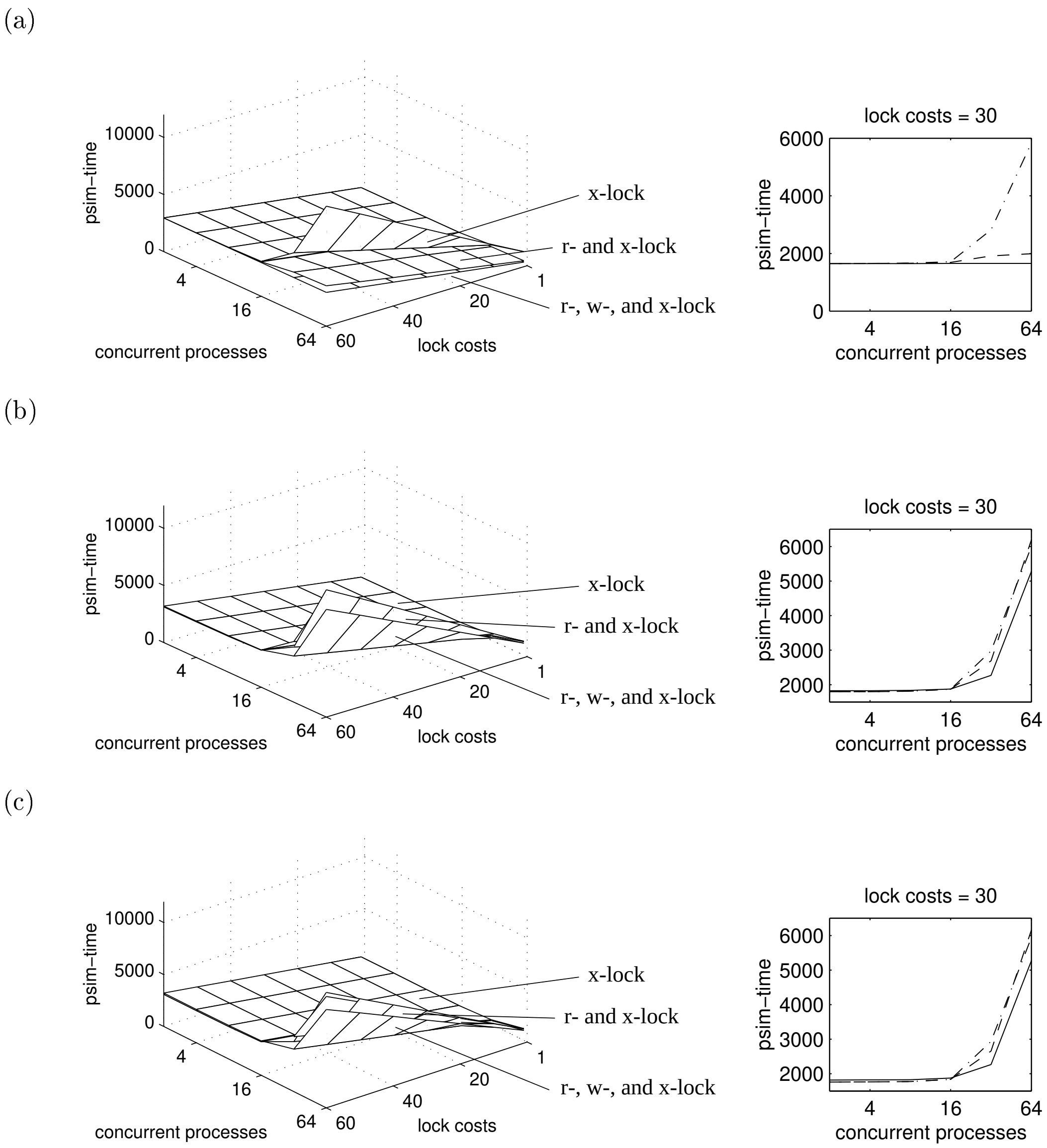
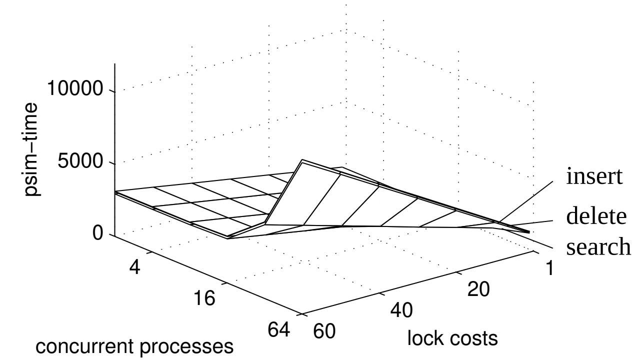
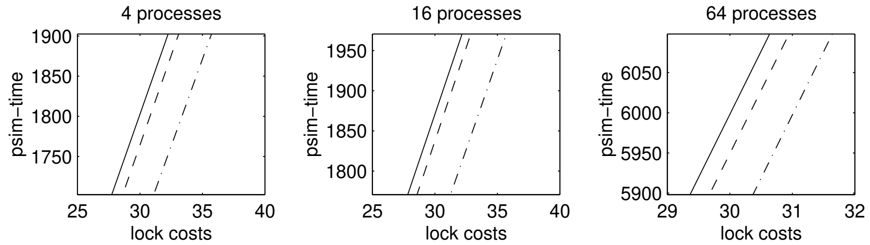


Figure 21: Average time needed to perform a *dictionary operation*, if the supported lock modes vary. (a) Search (b) Insertion (c) Deletion
Legend: —— r-, w-, and x-lock, $\;$ − − r- and x-lock, $\;$ − · − x-lock

(a)



(b)



Legend: ——— insert, − − delete, − · − search

(c)

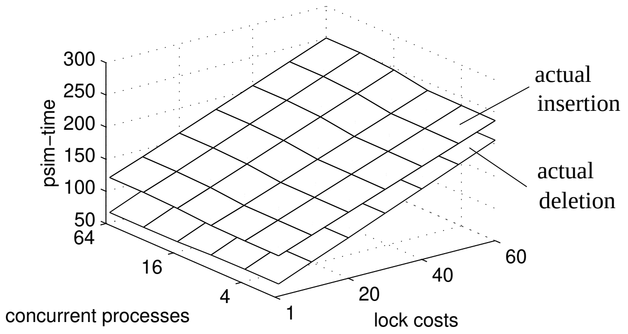
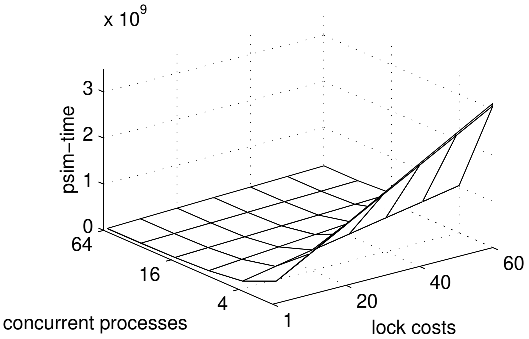


Figure 22: Comparison of the dictionary operations, if only x-locks are available (exemplary shown by the relaxed red-black tree).

(a)



(b)

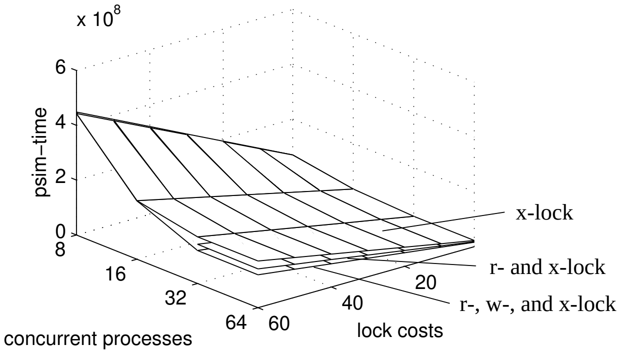


Figure 23: *Total time* needed in average to perform 1 000 000 dictionary operations.

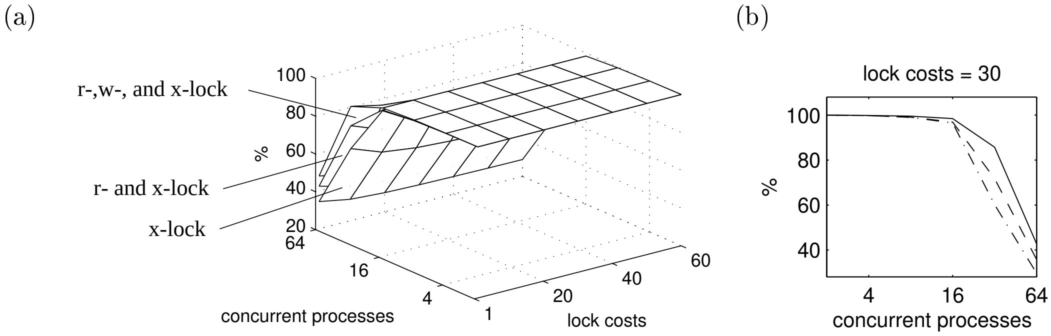


Figure 24: *Utilization* of the user processes, if the supported lock modes vary.
Legend: —— r-, w-, and x-lock, $\;-\;-$ r- and x-lock, $\;-\cdot-$ x-lock

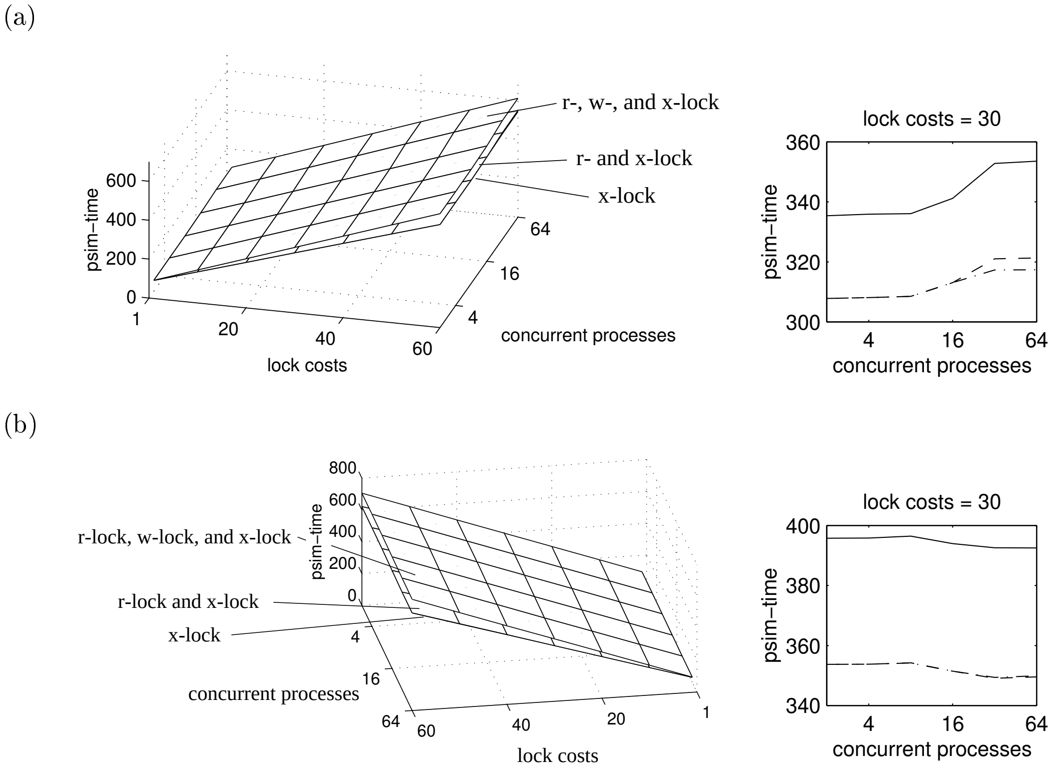


Figure 25: Average time needed to *handle a balance conflict*, if the supported lock modes vary. (exemplary shown by the chromatic tree) (a) red-balancing transformation (b) weight-balancing transformation *Legend:* —— r-, w-, and x-lock, $\;-\;-$ r- and x-lock, $\;-\cdot-$ x-lock

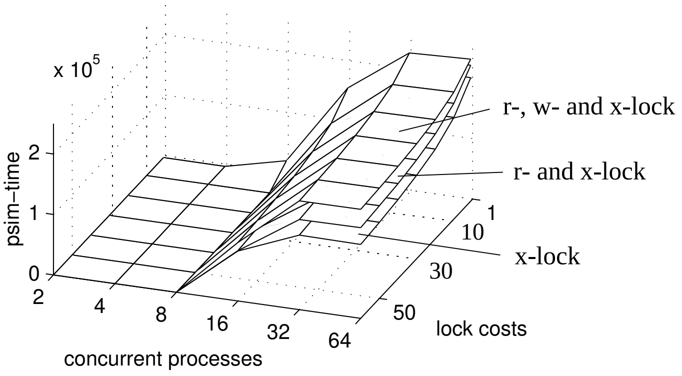


Figure 26: Number of unsettled rebalancing requests after performing 1 000 000 dictionary operations, if the supported lock modes vary (exemplary shown by the chromatic tree).

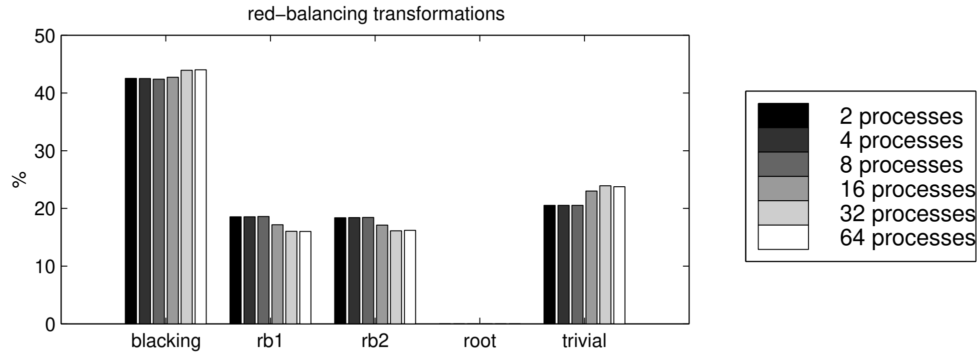


Figure 27: Proportion of the red-balancing transformations, if the second strategy to handle red-red conflicts is used (exemplary shown by the chromatic tree).

to perform a red-balancing transformation. We found that the second strategy is more suitable than the first one in order to keep the tree in balance, especially if only few rebalancing processes work for many user processes. Using the second strategy, the average time needed to perform a red-balancing transformation is considerably lower than using the first strategy, cf. Figure 28 (a). This is caused by the fact, that the proportion of the trivial case to the other red-balancing transformations increases, cf. Figure 28 (b) (or respectively compare Figure 9 with Figure 27). Therefore, during the same time more rebalancing transformations can be carried out, cf. Figure 28 (c), and finally the tree is better balanced, cf. Figure 28 (d). However, this has hardly an influence on the average time needed to perform a dictionary operation: the results of the first and the second set of experiments are almost identical.

Furthermore, by using the second strategy to handle red-red conflicts the number of failures to handle an overweight conflict in a basic relaxed balanced red-black tree is clearly reduced. Using the first strategy, the rate of failure was at most 2.7% of all overweight handlings. Using the second strategy, the handling of an overweight conflict almost never failed.

Analogously as for the first strategy we studied what changes, if less kinds of lock modes are supported. We found again that the second strategy to handle red-red conflicts has a clear advantage over the first one.

This result cannot be transfered to the handling of overweight conflicts. The same strategy applied to the weight-push transformation did not lead to a remarkable improvement.

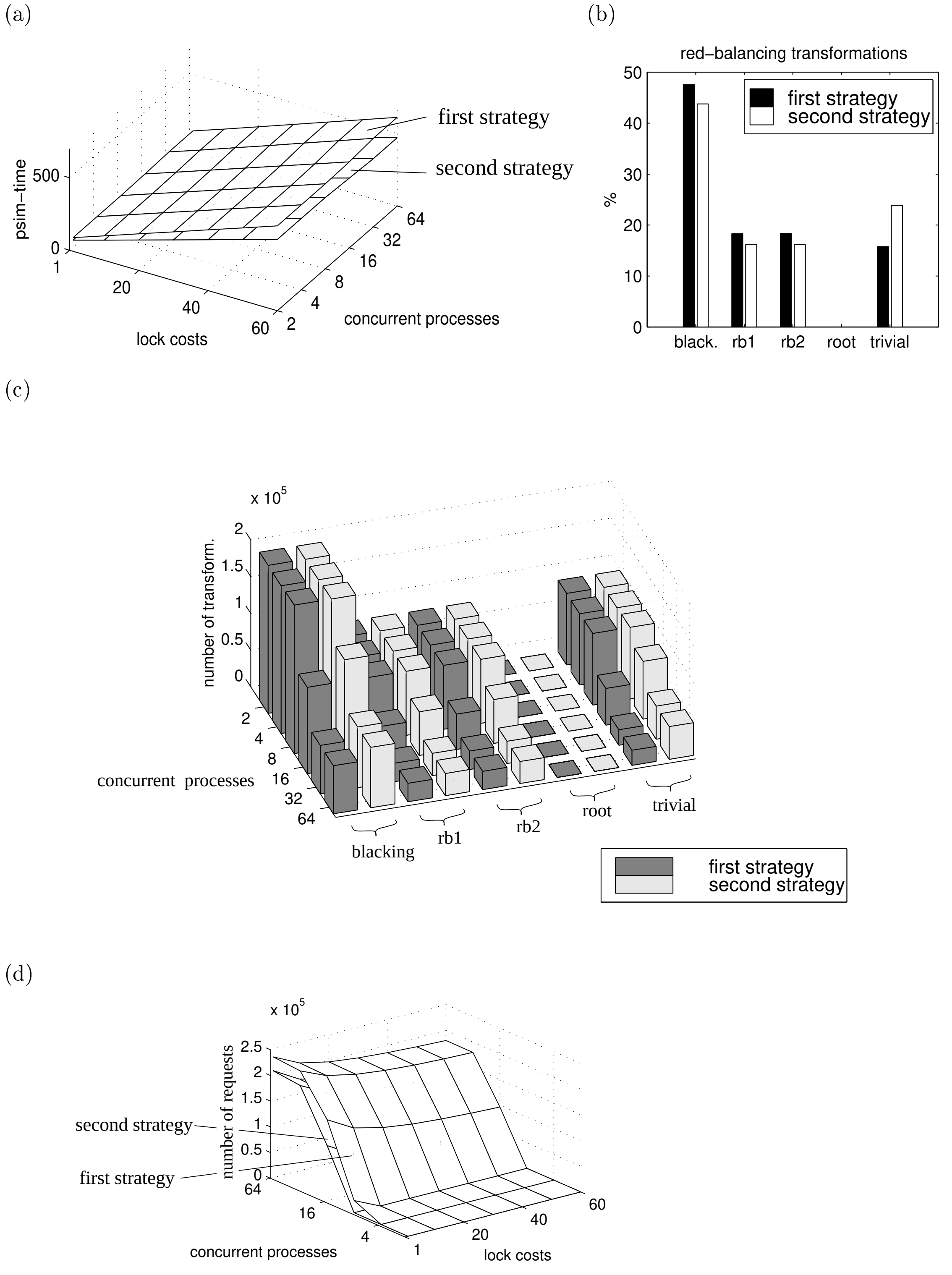


Figure 28: Comparison of the two strategies to handle red-red conflicts, exemplary shown by the chromatic tree: (a) Average time needed to perform a red-balancing transformation. (b) Proportion of the performed red-balancing transformations (using 63 user processes and 1 rebalancing process). (c) Total number of performed red-balancing transformations. (d) Number of unsettled rebalancing requests after the time used to perform 1000000 dictionary operations.

5 Conclusions

We have experimentally studied the performance of three relaxed balancing algorithms for red-black trees, the chromatic tree [5, 6, 26, 27], the basic relaxed balancing algorithm [11], and the relaxed red-black tree [17], and we compared them with the performance of the standard red-black tree [31] using the locking scheme of Ellis [7]. We have found that in a concurrent environment the relaxed balancing algorithms have a considerably higher performance than the standard algorithm. A comparison of the relaxed balancing algorithms among themselves yielded that with regard to the average response time of the dictionary operations the performance of the three relaxed balancing algorithms is comparably good. This gives grounds for the assumption that the general method of how to make strict balancing schemes relaxed generates satisfactory relaxed balancing algorithms, if it is applied to other classes of search trees as well.

Regarding the quality of the rebalancing there are differences between the three algorithms. We found that the chromatic tree has always the best performance. The rebalancing performance of the basic relaxed balancing algorithm is considerably lower than the performance of the chromatic tree but still satisfactory. The rebalancing of a relaxed red-black tree came off to be very complicated, since the restriction that a weight-temp operation cannot be applied, if a particular child of the red sibling of the overweighted node is also overweighted, led to a considerable high rate of failure. Furthermore, the hoped speed up of the weight-balancing is only insignificant, since only in less than 1.5% of all overweight handlings one of the larger weight-balancing transformations applies.

Furthermore, for all three relaxed balancing algorithms we have experimentally shown that the stategy to handle a newly created red-red conflict next after performing a blacking transformation has a significant advantage over an arbitrary order to handle red-red conflicts.

Acknowledgements

I would like to express my thanks to Kenneth Oksanen, who designed and implemented the first prototypes of the interpreter *psim* and thereby made the basis that enabled the experimental comparison of concurrent algorithms. I would also like to thank Eljas Soisalon-Soininen and Thomas Ottmann, who have decicively contributed to the origin of this work.

References

[1] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall International Editions, 1989.

[2] T. Anderson. The performance implications of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[3] L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Concurrent rebalancing of AVL trees: A fine-grained approach. In *Proceedings of the 3th Annual European Conference on Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 321–429, 1997.

[4] L. Bougé, J. Gabarró, X. Messeguer, and N. Schabanel. Concurrent rebalancing of AVL trees: A fine-grained approach. Technical Report RR97-13, LIP, ENS Lyon, France, 1997.

[5] J. Boyar, R. Fagerberg, and K. Larsen. Amortization results for chromatic search trees, with an application to priority queues. In *4th International Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, pages 270–281, 1995.

[6] J. Boyar and K. Larsen. Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences*, 49:667–682, 1994.

[7] C.S. Ellis. Concurrent search in AVL-trees. *IEEE Transactions on Computers*, C-29(29):811–817, 1980.

[8] J. Gabarró, X. Messeguer, and D. Riu. Concurrent rebalancing on hyperred-black trees. In *Proceedings of the 17th International Conference of the Chilean Computer Science Society*, pages 93–104. IEEE Computer Society Press, 1997.

[9] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[10] S. Hanke. Chromatic search trees revisited. Technical Report 90, Institut für Informatik, Universität Freiburg, Germany, 1997.

[11] S. Hanke, T. Ottmann, and E. Soisalon-Soininen. Relaxed balanced red-black trees. In *Proc. 3rd Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science*, volume 1203, pages 193–204, 1997.

[12] T. Johnson and D. Shasha. A framework of the performance analysis of concurrent b-tree algorithms. *ACM Symposium on Principles of Database Systems*, pages 273–287, 1990.

[13] T. Johnson and D. Shasha. The performance of current B-tree algorithms. *ACM Transactions on Database Systems*, 18(1):51–101, March 1993.

[14] J.L.W. Kessels. On-the-fly optimization of data structures. *Communications of the ACM*, 26:895–901, 1983.

[15] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *1986 Fall Joint Computer Conference*, pages 380–389, 1986.

[16] K. Larsen. AVL trees with relaxed balance. In *Proc. 8th International Parallel Processing Symposium*, pages 888–893. IEEE Computer Society Press, 1994.

[17] K. Larsen. Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica*, 35(10):859–874, 1998.

[18] K. Larsen and R. Fagerberg. B-trees with relaxed balance. In *Proc. 9th International Parallel Processing Symposium*, pages 196–202. IEEE Computer Society Press, 1995.

[19] K. Larsen, T. Ottmann, and E. Soisalon-Soininen. Relaxed balance for search trees with local rebalancing. *Fifth Annual European Symposium on Algorithms, Lecture Notes in Computer Science*, (1284):350–363, 1997.

[20] K. Larsen, E. Soisalon-Soininen, and P. Widmayer. Relaxed balance through standard rotations. In *Workshop on Algorithms and Data Structures*, Halifax, Nova Scotia, Canada, August 1997.

[21] P. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[22] B.-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proc. 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, 1994.

[23] L. Malmi. A new method for updating and rebalancing tree-type main memory dictionaries. *Nordic Journal of Computing*, 3:111–130, 1996.

[24] L. Malmi. *On Updating and Balancing Relaxed Balanced Search Trees in Main Memory.* PhD thesis, Helsinki University of Technology, 1997.

[25] X. Messeguer and B. Valles. Hyperchromatic trees: a fine-grained approach to distributed algorithms on redblack trees. Technical Report LSI-98-13-R, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, 1998.

[26] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proc. 10tm ACM Symposium on Priciples of Database Systems*, pages 192–198, 1991.

[27] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica 33*, pages 547–557, 1996.

[28] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proc. 6th ACM Symposium on Principles of Database Systems*, pages 170–176, 1987.

[29] T. Ottmann and E. Soisalon-Soininen. Relaxed balancing made simple. Technical Report 71, Institut für Informatik, Universität Freiburg, Germany, 1995.

[30] Y. Sagiv. Concurrent operations on b*-trees with overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, 1986.

[31] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[32] E. Soisalon-Soininen and P. Widmayer. Relaxed balancing in search trees. In D.-Z.Du and K.-I. Ko, editors, *Advances in Algorithms, Languages, and Complexity: Essays in Honor of Ronald V. Book*. Kluwer Academic Publishers, Dordrecht, 1997.

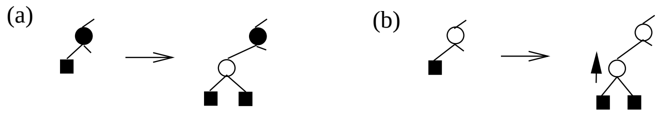
[33] V. Srinivasan and M.J. Carey. Performance of B-tree concurrency control algorithms. *Proc. ACM International Conference on Management of Data*, pages 416–425, 1991.

Appendix

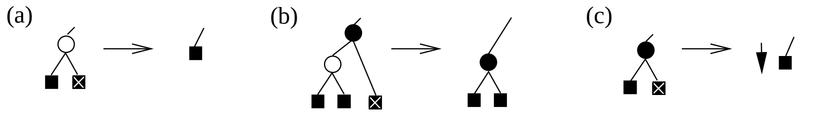
A Operations of the Standard Red-Black Tree [31]

Squares denote leaves, circles denote general nodes, i.e. either internal nodes or leaves. Filled nodes denote black nodes and nodes without fill denote red nodes. Half filled nodes denote nodes that are either black or red. "↑" denotes the call of the rebalancing procedure in Red-Black Tree Insertion, "↓" denotes the call of the rebalancing procedure in Red-Black Tree Deletion, and "×" denotes the leaf that should be removed. Symmetric cases are ommitted.

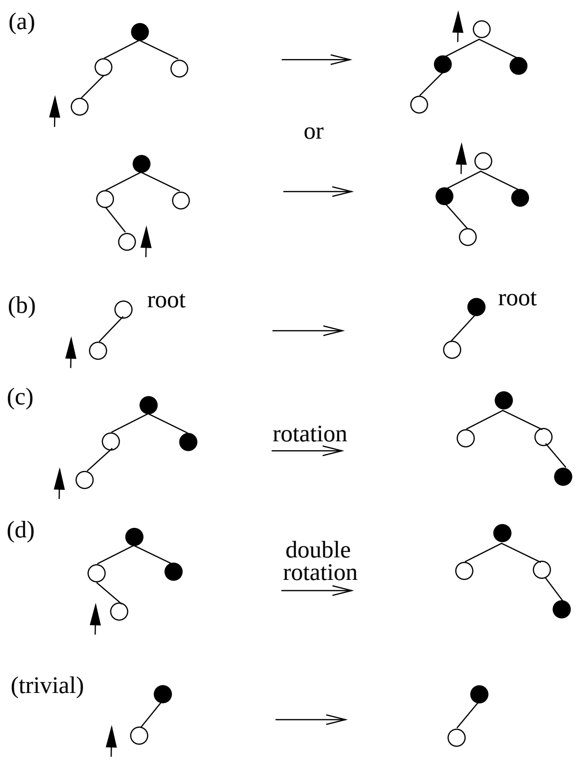
Insertion of a new item and call of the rebalancing procedure in Red-Black Tree Insertion:



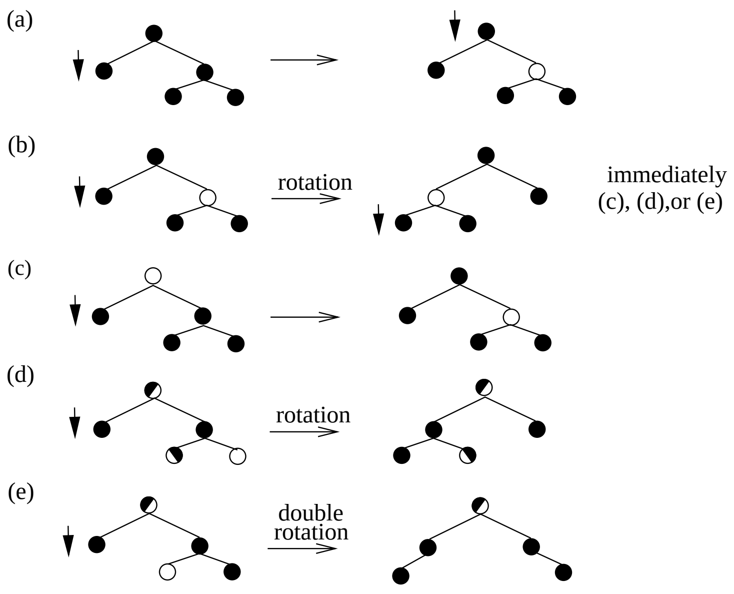
Deletion of an item and call of the rebalancing procedure in Red-Black Tree Deletion:



Local structural changes by the rebalancing procedure in Red-Black Tree Insertion:



Local structural changes by the rebalancing procedure in Red-Black Tree Deletion:

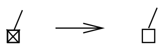


B Operations of the Basic Relaxed Balancing Algorithm [11]

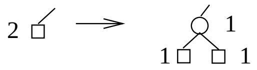
Squares denote leaves, circles denote general nodes, i.e. either internal nodes or leaves, and the labels denote weights $\in \{0, 1, 2\}$, where $0 =$ red, $1 =$ black, and $2 =$ black with up-out request. "$\uparrow$" denotes an up-in request and "$\times$" denotes a removal request.
Symmetric cases are omitted.

(1) Insertion and (2) Deletion of an item in a relaxed balanced red-black tree:

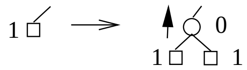
(1.a)



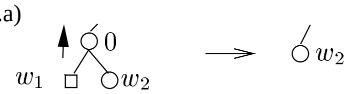
(1.b)



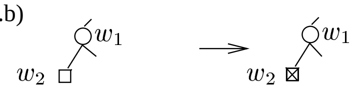
(1.c)



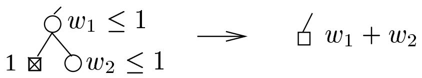
(2.a)



(2.b)

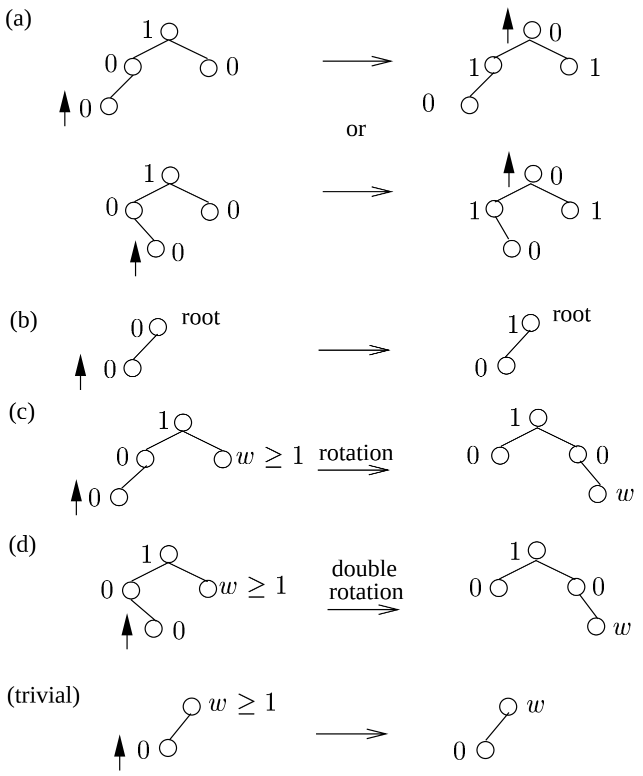


Handling a removal request:

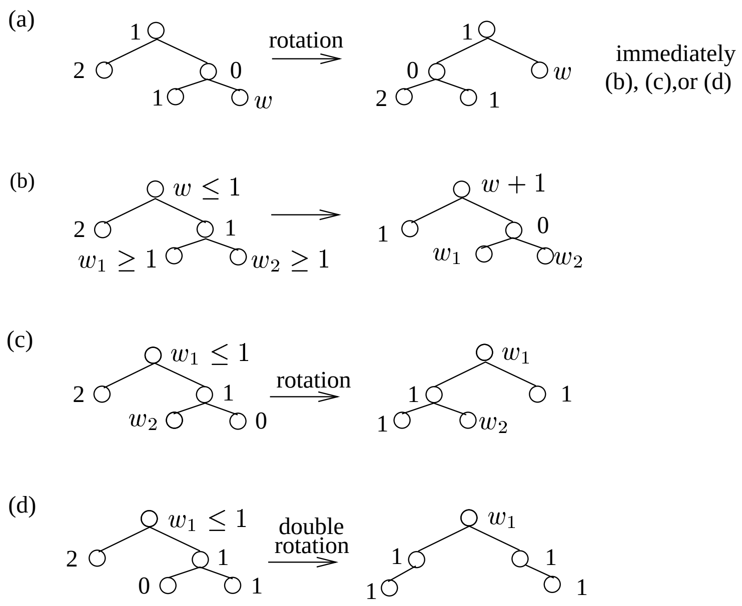


The following rebalancing operations are exactly the same as in the strict case:

Handling an up-in request:



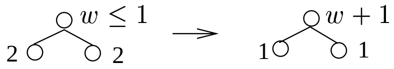
Handling an up-out request:



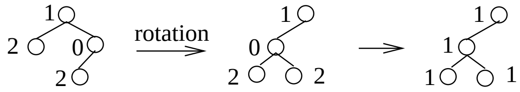
(Note that (b) implements the rebalancing transformations (a) and (c) in Red-Black Tree Deletion of the strict case, cf. Appendix A.)

The additional rebalancing operations to handle an up-out request:

(e)



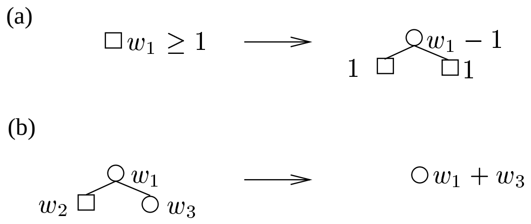
(a+e)



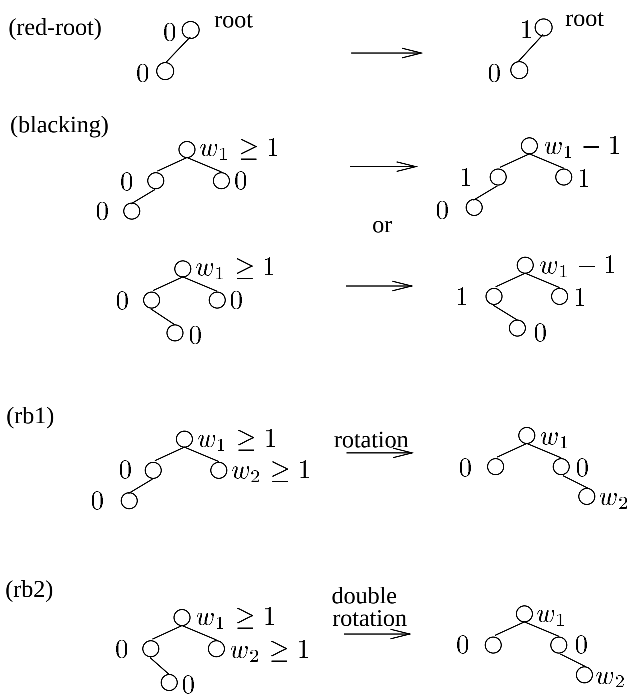
C Operations of the Chromatic Tree [6]

Squares denote leaves, circles denote general nodes, i.e. either internal nodes or leaves, and the labels denote weights: 0 = red, 1 = black, and > 1 is overweighted.
Symmetric cases are omitted.

(a) Insertion and (b) Deletion of an item in a chromatic tree:

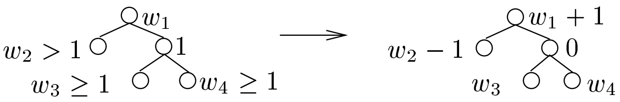


Handling a red-red conflict:

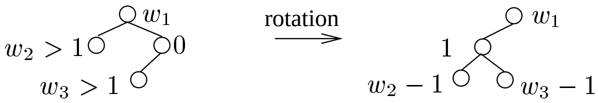


Handling an overweight conflict:

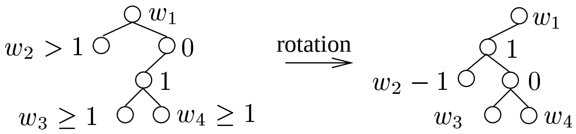
(push)



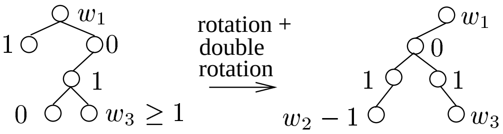
(w1)



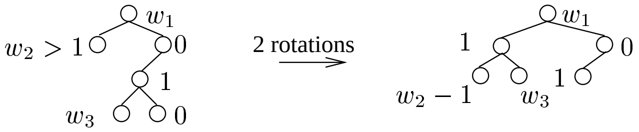
(w2)



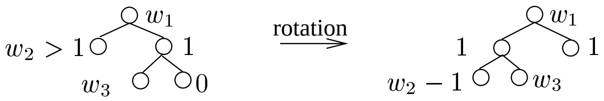
(w3)



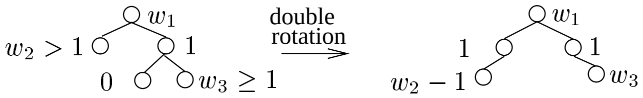
(w4)



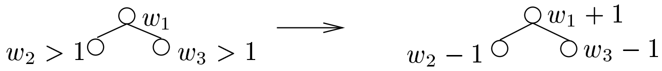
(w5)



(w6)



(w7)

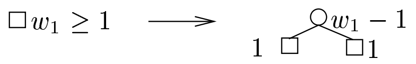


D Operations of the Relaxed Red-Black Tree [17]

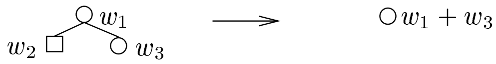
Squares denote leaves, circles denote general nodes, i.e. either internal nodes or leaves, and the labels denote weights: 0 = red, 1 = black, and > 1 is overweighted.
Symmetric cases are omitted.

(a) Insertion and (b) Deletion of an item in a chromatic tree:

(a)

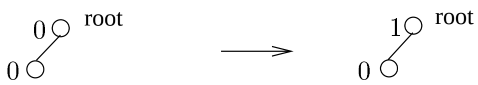


(b)

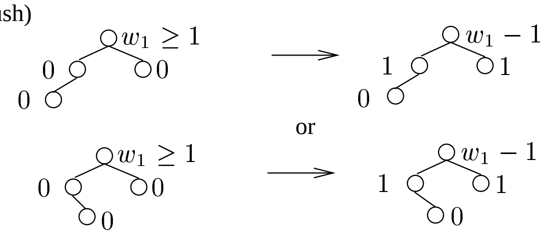


Handling a red-red conflict:

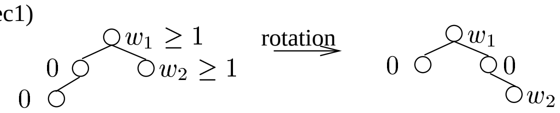
(red-root)



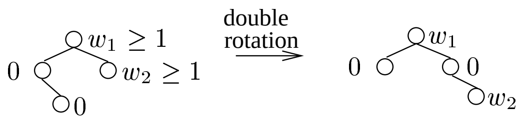
(red-push)



(red-dec1)

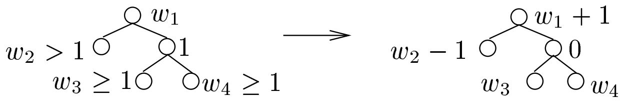


(red-dec2)

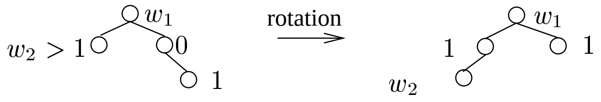


Handling an overweight conflict:

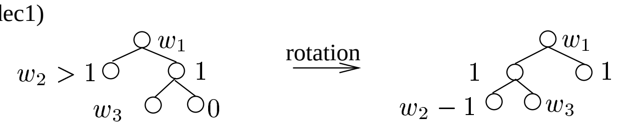
(weight-push)



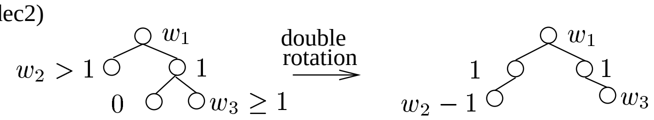
(weight-temp)



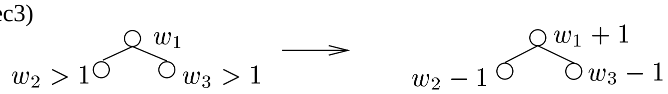
(weight-dec1)



(weight-dec2)



(weight-dec3)



E Comparison of the Rebalancing Operations

The following table compares the rebalancing operations of the Algorithms. Each row corresponds to one rebalancing transformation and lists the different names of it. Furthermore, the number of structural changes is listed that is needed to perform the transformation. The last column says, whether the rebalancing operations resolves at least one red-red conflict or one unit of overweight. Since the weights of a standard red-black tree and the basic relaxed balanced red-black tree are bounded by two, we identify two rebalancing transformations, if one is a generalization of the other with regard to the overweight.

Standard Red-Black Tree [31]	**Basic Relaxed Balancing Algorithm** [11]	**Chromatic Tree** [6]	**Relaxed Red-Black Tree** [17]	number of structural changes	$\not\uparrow, \not\downarrow$
(Insert a)	(up-in a)	(blacking)	(red-push)	—	yes
(Insert b)	(up-in b)	(red-root)	(red-root)	—	yes
(Insert c)	(up-in c)	(rb1)	(red-dec1)	1 rotation	yes
(Insert d)	(up-in d)	(rb2)	(red-dec2)	1 double rotation	yes
(Delete a) resp. (Delete c)	(up-out b)	(push)	(weight-push)	—	yes
(Delete d)	(up-out c)	(w5)	(weight-dec1)	1 rotation	yes
(Delete e)	(up-out d)	(w6)	(weight-dec2)	1 double rotation	yes
—	(up-out e)	(w7)	(weight-dec3)	—	yes
(Delete b+c)	(up-out a+b)	(w2)	—	1 rotation	yes
(Delete b+d)	(up-out a+c)	—	—	2 rotations	yes
—	—	(w4) resolves same situation as (up-out a+c)	—	1 double rotation	yes
(Delete b+e)	(up-out a+d)	(w3)	—	1 rotation + 1 double rotation	yes
—	(up-out a+e)	(w1)	—	1 rotation	yes
first part (Delete b) of (b+c), (b+d), and (b+e)	first part (up-out a) of (a+b), (a+c), (a+d), and (a+f)	first part of (w1), (w2), and (w3)	(weight-temp)	1 rotation	no

S. Hanke: The Performance of Concurrent Red-Black Tree Algorithms, Technical Report 115, Institut für Informatik, Universität Freiburg, Germany, 1998

Errata

Section 2.5, page 8:

If exactly the same situation occurs in a chromatic tree or relaxed red-black tree, then the *red-balancing* transformation can be carried out immediately.

Section 3.1, page 10:

Therefore, every *node that has a* rebalancing request is represented by an item of a problem queue.

Section 4.2, page 13:

As key space we choose the interval [0,*2 000 000 000*].

Section 4.2.1, page 15:

For all three algorithms the average *length of a search path* to a leaf is near the minimum of $\log_2 n$: ...

Figure 26, page 30:

The label of the z-axis must be "*number of requests*".