The Performance of Concurrent Red-Black Tree Algorithms

Sabine Hanke

Institut für Informatik, Universität Freiburg
Am Flughafen 17, D-79110 Freiburg, Germany
hanke@informatik.uni-freiburg.de, fax: ++49 +761 203 8162

Abstract. Relaxed balancing has become a commonly used concept in the design of concurrent search tree algorithms. 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 the strictly balanced red-black tree and three relaxed balancing algorithms for red-black trees using the simulation of a multi-processor machine.

1 Introduction

In some applications—like real-time embedded systems as the switching system of mobile telephones—searching and updating a dictionary must be extremely fast. Dictionaries stored in main-memory are needed which can be queried and modified concurrently by several processes. Standard implementations of main-memory dictionaries are balanced search trees like red-black trees or AVL-trees. If they are implemented in a concurrent environment, there must be a way to prevent simultaneous reading and writing of the same parts of the data structure. A common strategy for concurrency control is to lock the critical parts. Only a small part of the tree should be locked at a time in order to allow a high degree of concurrency. For an efficient solution to the concurrency control problem, the concept of relaxed balancing is used in the design of many recent concurrent search tree algorithms (see [17] for a survey). *Relaxed balancing* means that the rebalancing is uncoupled from the update and may be arbitrarily delayed and interleaved.

With regard to the comparison of relaxed balancing algorithms with other concurrent search tree algorithms and an experimental performance analysis, the B-tree is the only algorithm which has been studied well. 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 [9,18]. Furthermore, the efficiency of group-updates in relaxed balanced B-trees is experimentally analyzed [15]. So far, the performance of concurrent relaxed balanced binary search trees has hardly been studied by experiments. None of the experimental research known to us examines relaxed balanced binary trees in concurrent environments.

J.S. Vitter and C.D. Zaroliagis (Eds.): WAE'99, LNCS 1668, pp. 286–300, 1999.

Bougé et al. [1] and Gabarró et al. [5] perform some serial experiments in order to study the practical worst-case and average convergence time of their relaxed balancing schemes. Malmi presents sequential implementation experiments applying a periodical rebalancing algorithm to the chromatic tree [11] and the height-valued tree [12]. The efficiency of group-operations is studied in height-valued trees [13]. It is shown that group updates provide a method of considerably speeding up search and update time per key [13].

This paper presents an experimental comparison of the strictly balanced red-black tree and three relaxed balanced red-black trees in a concurrent environment. We examine the algorithm proposed by Sarnak and Tarjan [16] combined with the locking scheme of Ellis [4], the chromatic tree [2], the algorithm that is obtained by applying the general method of making strict balancing schemes relaxed to red-black trees [8], and Larsen's relaxed red-black tree [10]. The three relaxed balancing algorithms are relatively similar and have the same analytical bounds on the number of needed rebalancing transformations. The chromatic tree and Larsen's relaxed red-black tree can both be seen as tuned versions of the algorithm obtained 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 more grained, so that fewer nodes at a time must be locked. But whether, in practice, the relaxed red-black tree really fits better is unknown.

In order to clarify these points, we implemented the four algorithms using the simulation of a multi-processor machine, and we compared the performance of the algorithm in highly dynamic applications of large data. 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 we first give a short review of the red-black tree algorithms; then we present the relief algorithms that are necessary to implement the algorithms in a concurrent environment. In Section 3 we describe the performed experiments and the results.

2 Concurrent Red-Black Tree Algorithms

2.1 The Rebalancing Schemes

In this section we give a short review of the four red-black tree algorithms that we want to compare. For each of them, the number of structural changes after an update is bounded by $O(1)$ and the number of color changes by $O(\log n)$, if n is the size of the tree. The rebalancing is amortized constant [2,8,10,16].

The Standard Red-Black Tree In the following, we consider the variant of red-black trees by Sarnak and Tarjan [16]. Deviating from the original proposal by Guibas and Sedgewick [6], here the nodes, and not the edges, are colored red or black. Since the relaxed balanced red-black trees described below can all be seen as relaxed extensions of the strict rebalancing algorithm by Sarnak and Tarjan, in the following we call it the *standard red-black tree*.

The balance conditions of a standard red-black tree require that:

1. each 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.

Inserting a key may introduce two adjacent red nodes on a path. This is called a *red-red conflict*. Immediately after generating a red node p that has a red parent, the insert operation performs the rebalancing. If p's parent has a red sibling, then the red-red conflict is handled by color changes. This transformation may produce a new violation of the red constraint at the grandparent of p. 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.

Deleting a key may remove a black node from a path. This violation of the black constraint is handled immediately by the delete operation. Using only color changes, it is bubbled up in the tree until it can be finally resolved by an appropriate transformation.

The Basic Relaxed Balancing Algorithm The nodes of a *basic relaxed balanced red-black tree* [8] 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*. The relaxed balance conditions require that:

1. the sum of the number of black nodes plus the number of up-out requests is the same on each path from the root to a leaf,
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 contrast to the standard red-black tree, a red-red conflict is not resolved immediately after the actual insertion. The insert operation only marks a violation of the red constraint by an up-in request. In general, deleting a key leads to a removal request only, where the actual removal is part of the rebalancing. The handling of a removal request may remove a black node from a path. This is marked by an up-out request.

The task of rebalancing a basic relaxed balanced red-black tree involves handling all up-in, removal, and up-out requests. For this the rebalancing operations of the standard red-black tree are used, which are split into small restructuring

steps that can be carried out independently and 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 a top-down manner if they meet in the same area; otherwise, in any arbitrary order.

Furthermore, in order to keep the synchronization areas as small as possible, two additional rebalancing transformations are defined that apply to situations that cannot occur in a strictly balanced red-black tree.

Chromatic Tree The nodes of a *chromatic tree* [14] are colored red, black, or overweighted. The color of a node p is represented by an integer $w(p)$, 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. The relaxed balance conditions are:

1. all leaves are not red and
2. the sum of weights on each path from the root to a leaf is the same.

Analogous to the basic relaxed red-black tree, the rebalancing is uncoupled from the updates. Here the delete operation performs the actual removal. A deletion may remove a non-red node and increment the weight of a node on the path instead. Thereby, the node may become overweighted. This is called an *overweight conflict*.

The task of rebalancing a chromatic tree involves removing all red-red and overweight conflicts from the tree. For this purpose, Nurmi and Soisalon-Soininen propose a set of rebalancing operations [14]. A more efficient set of transformations is defined by Boyar and Larsen [2,3].

Several red-red conflicts at consecutive nodes must be handled in a top-down manner. Otherwise, the rebalancing transformations can be applied in any arbitrary order.

The basic relaxed balanced red-black tree differs from the chromatic tree mainly in that deletions are accumulated by removal requests. However, an up-out request is equivalent to one unit of overweight. Therefore, a basic relaxed balanced red-black tree always fulfills the balance conditions of chromatic trees. It differs from an arbitrary chromatic tree only in that overweights are bounded by two. Furthermore, a detailed comparison of the rebalancing transformations of both algorithms depicts that the transformations of the chromatic tree can be seen as generalizations of the transformations of the basic relaxed balancing algorithm.

Larsen's Relaxed Red-Black Tree The relaxed balance conditions, the operations insert and delete, and the transformations to handle a red-red conflict of *Larsen's relaxed red-black tree* [10] are exactly the same as those of chromatic trees. The set of rebalancing transformations to handle an overweight conflict is reduced to a smaller collection of operations consisting of only six operations

instead of nine. Five of them, which perform at most one rotation or double rotation, are identical with operations of the chromatic tree. The sixth transformation, *weight-temp*, performs one rotation but does not resolve an overweight conflict. If we assume that weight-temp is followed immediately by one of the other transformations, then this combined transformation is almost identical to one of the remaining operations of the chromatic tree. But weight-temp can also be carried out independently and interleaved freely with other operations. This has the advantage that fewer nodes must be locked at a time. On the other hand, in order to prevent weight-temp from generating its symmetric case, it introduces a restriction in which order weight-balancing transformations can be carried out.

2.2 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 administration of the rebalancing requests and a suitable mechanism for concurrency control.

Administration of the Rebalancing Requests In the literature, there are three different suggestions for the way in which rebalancing processes can locate rebalancing requests in the tree. One idea is to traverse the data structure randomly in order to search for rebalancing requests [14]. Another idea is to mark all nodes on the path from the root to a leaf where an update occurs, so that a rebalancing process can restrict its search to the marked subtrees [11]. The third suggestion is to use a problem queue [3], which has the advantage that rebalancing processes can search purposefully for imbalance situations. Here it is necessary for each node to have a parent pointer in addition to the left and right child pointers. Since the problem queue is mentioned only very vaguely [3], in the following we propose how to maintain a queue to administrate the rebalancing requests.

In addition to the color field of a node, each rebalancing request is implemented by a pointer to the node, the *request pointer*. Whenever a rebalancing request is generated, the corresponding request pointer is placed in a problem queue.

Since it is possible to remove nodes from the tree which have rebalancing requests, the problem queue may contain request pointers that are no longer valid. In order to avoid side effects, every node should be represented by only one pointer in a queue. This can be guaranteed by using a control bit for each node that is set if a request pointer is placed in a problem queue.

If a node for which a request pointer exists is deleted from the tree, then after the removal the node must be marked explicitly as removed. A rebalancing process following the link of the request pointer is now able to notice that the node is no longer in the tree. Freeing the memory allocated for the node is delayed and becomes part of the rebalancing task.

Concurrency Control A suitable mechanism for concurrency control in strictly balanced search trees is the locking protocol by Ellis [4]. Nurmi and Soisalon-Soininen propose a modification of Ellis's locking scheme for relaxed balanced search trees [14]. In the following, we only mention the features of both locking schemes which are the most important and which are relevant to this paper.

Both protocols use 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 or w-locked.

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.

Using the locking protocol by Ellis, a process that performs an update operation w-locks the whole path. This is necessary in order to perform the rebalancing immediately after the update. In the locking protocol for relaxed balanced search trees, it is sufficient only to use w-lock coupling during the search phase of the update operation.

A rebalancing process 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. Structural changes are implemented by exchanging the contents of nodes.

Nurmi and Soisalon-Soininen suggest that rebalancing processes locate rebalancing requests by traversing the tree nondeterministically, so that rebalancing processes can always w-lock the nodes incipient with the top-most one. This guarantees that the locking scheme is dead-lock free [14].

If we use a problem queue in order to administrate the rebalancing requests, the situation is different. The rebalancing transformations of red-black trees 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. Thus, if a rebalancer follows the link of a request pointer when searching for an imbalance situation in the tree, it cannot lock the top-most node involved first. Therefore, we extend the locking scheme for relaxed balanced search trees in the following way.

A rebalancing process always only tries to w-lock ancestor nodes. If the parent of a node is still w-locked or x-locked by another process, then in order to avoid a dead-lock situation, the rebalancer immediately releases all locks it holds. Since all locks that may block the process are taken in top-down direction, dead-lock situations are avoided. The results of our experiments, which we present in the following section, 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.

Normally the w-lock of each node, the contents of which will be changed, is converted to an x-lock. However, this conversion is not necessary if only the

parent pointer of the node is to be updated. This is because rebalancing processes are the only processes that use parent pointers.

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

3 The Experiments

In order to compare the concurrent red-black tree algorithms, we have used the simulation of a multi-processor environment psim [7] that was developed in cooperation with Kenneth Oksanen from the Helsinki University of Technology. In the following, we give a short description of the simulator psim. Then we describe the experiments and briefly present the main results.

3.1 Simulation of a Concurrent Environment

The interpreter psim serves to simulate algorithms on parallel processors [7]. The algorithms are formulated by using psim's own high-level programming language which can be interpreted, executed, and finally statistically analyzed.

psim simulates an abstract model of a multi-processor machine, a CRCW-MIMD machine with shared constant access time memory. 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 at the same time. 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.) also takes one unit of psim-time. The access of global data can be synchronized by using 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 lock operation is constant for each and can be chosen by the user. The overhead of lock contention is not modeled.

3.2 Experiments

Our aim is to study the behavior of the red-black tree algorithms in highly dynamic applications of large data. For this, we implemented the algorithms in the psim programming language and concurrently performed 1 000 000 dictionary operations (insertions, deletions, and searches) on an initially balanced standard red-black tree, which was built up by inserting 1 000 000 randomly chosen keys into an empty tree. As key space we choose the interval $[0, 2\,000\,000\,000]$. During

the experiments, we varied the number of processes and the time that is needed to perform a lock operation. In the case of strict balancing, 2, 4, 8, 16, 32, or 64 user processes apply concurrently altogether 1 000 000 random dictionary operations to the tree. In the case of relaxed balancing, either 1, 3, 7, 15, 31, or 63 user processes are used, plus a single rebalancing process, which gets its work from a FIFO problem queue. Analogous to [9], the probabilities that a dictionary operation is an insert, delete, or search operation are $p_i = 0.5$, $p_d = 0.2$, and $p_s = 0.3$. The costs of an arbitrary lock operation—excluding the time waiting to acquire a lock, if it is already taken by another process—are constant: either 1, 10, 20, 30, 40, 50, or 60 units of psim-time, motivated by measurements of lock costs that we performed on real machines.

Comparison of Strict and Relaxed Balancing In the following, we confirm by experiment that relaxed balancing has a significant advantage over strict balancing in a concurrent environment. For this, we compare the performance of the standard red-black tree with that of the chromatic tree.

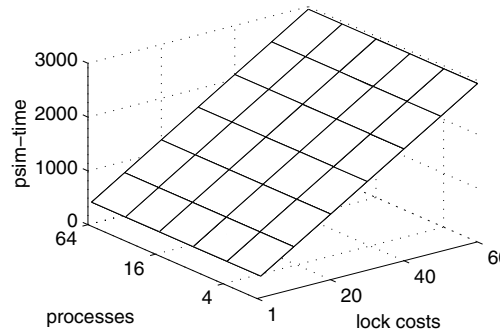
The search is the only dictionary operation in which the standard red-black tree supplies almost the same results as the chromatic tree (cf. Figure 1.1). On average, a search is only 1.18 times faster in a chromatic tree than in a standard red-black tree. This is because both use r-lock coupling, and no rebalancing is needed after the search.

In the case of the update operations, the chromatic tree is considerably faster than the standard red-black tree (cf. Figure 1.2). The speed-up factor depends on the number of processes and the lock costs. It varies from a minimal factor of 1.21 (in the case of 2 processes and lock costs 60) to a maximal factor of 25.21 (in the case of 32 processes and lock costs 1, see Table 1).

The total time needed on average to perform 1 000 000 dictionary operations is depicted in Figure 2. Only in the case of two concurrent processes does the standard red-black tree terminate earlier than the chromatic tree. But note that, in this case, the dictionary operations are performed serially in the chromatic tree, and the rebalancing process is hardly used (cf. Table 3). If more than two processes work concurrently, then the chromatic tree always terminates earlier than the standard red-black tree. The higher the number of processes, the clearer the differences between both algorithms become. In the case of 64 processes, the chromatic tree is up to 20.22 times faster than the standard red-black tree (see Table 2). As Figure 3 shows, this is because an increase in the number of user processes in a standard red-black tree leads to almost no saving of time.

The utilization of the user processes reflects why the standard red-black tree performs so badly (cf. Figure 4). Since during an update operation, the whole path from the root to a leaf must be w-locked, the root of the tree becomes a bottleneck. Therefore, with an increasing number of concurrent processes, the utilization of the processes decreases in a steep curve towards only 4%. In contrast, the user processes that modify a chromatic tree use w-lock coupling during the search phase of an update operation. The rebalancing process locks only a small number of nodes at a time. Thus, for up to 16 concurrent processes, the

(1)



(2)

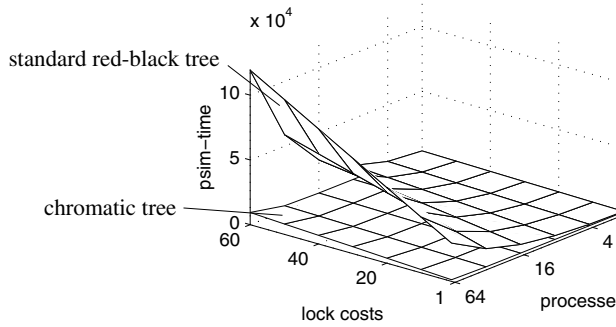


Fig. 1. Average time needed to perform (1) a search operation and (2) an update operation.

	number of processes					
lock costs	2	4	8	16	32	64
1	1.84	3.30	6.86	13.71	25.21	23.19
10	1.51	2.61	5.53	11.15	19.53	17.16
20	1.37	2.32	4.97	10.08	17.23	14.98
30	1.30	2.18	4.69	9.54	16.07	13.94
40	1.26	2.09	4.52	9.21	15.38	13.34
50	1.23	2.03	4.40	8.99	14.94	12.93
60	1.21	1.99	4.32	8.83	14.61	12.63

Table 1. Factor of speed-up when performing an update operation in a chromatic tree compared with a standard red-black tree.

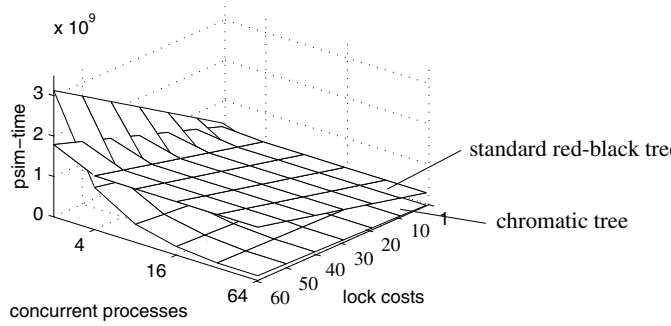


Fig. 2. Total time needed on average to perform 1 000 000 dictionary operations.

	number of processes					
lock costs	2	4	8	16	32	64
1	0.81	2.04	4.73	9.97	19.14	20.22
10	0.68	1.63	3.78	7.94	14.68	14.98
20	0.63	1.46	3.39	7.13	12.91	13.07
30	0.61	1.38	3.20	6.73	12.04	12.17
40	0.59	1.33	3.08	6.49	11.51	11.63
50	0.58	1.30	3.01	6.33	11.18	11.28
60	0.57	1.28	2.95	6.21	10.96	11.03

Table 2. Factor of speed-up when performing 1 000 000 dictionary operations in a chromatic tree compared with a standard red-black tree.

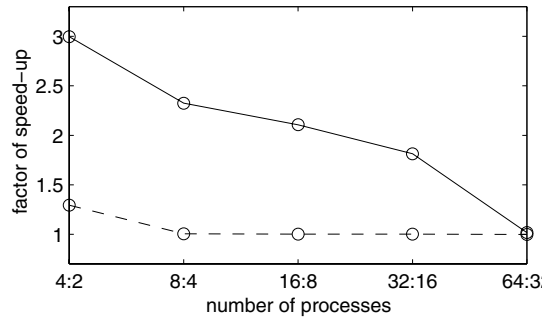


Fig. 3. Gain in time by increasing the number of processes. —— chromatic tree, – – standard red-black tree

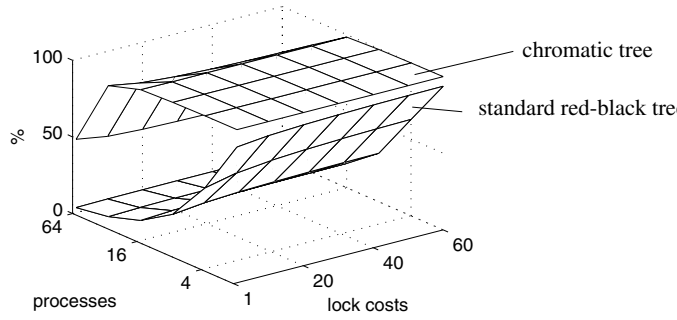


Fig. 4. Utilization of the user processes.

utilization of the user processes is always over 98%. By a further increase of the number of user processes, the utilization decreases clearly, but even in the case of 64 concurrent processes, it is still between 42% and 46% (cf. Figure 4).

Comparison of the Relaxed Balancing Algorithms In this section we experimentally compare the performance of the chromatic tree [2], Larsen's relaxed red-black tree [10], and the basic relaxed balanced red-black tree [8].

Regarding the average response time of the dictionary operations and the utilization of the user processes, the three relaxed balancing algorithms supply nearly identical results. Thus, in all three cases, almost the same time is needed on average to perform 1 000 000 dictionary operations.

The length of a path from the root to a leaf is at most 26, if two or four concurrent processes are used. In the case of eight concurrent processes, the maximum length of a path is 27, and in the case of 16, 32, and 64 processes, it is 28. The average length of a search path varies from 21.7 to 21.9. This increase is caused by the proportion of only one rebalancing process to a large number of user processes.

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

Regarding the quality of the rebalancing, there are differences between the algorithms. A measure of how well the tree is balanced is the size of the problem queue after performing the 1 000 000 dictionary operations. Figure 5 shows the number of unsettled rebalancing requests. The chromatic tree and Larsen's relaxed red-black tree are comparably well balanced. If eight or more concurrent processes are used, the basic relaxed balancing algorithm always leaves considerably more rebalancing requests than the other algorithms. This is caused by the additional work a rebalancer must do in a basic relaxed balanced red-black tree in order to settle removal requests as well. In total, about 20–40% more rebalancing requests are generated than in a chromatic tree.

Next we consider the capacity of the rebalancing process. Figure 6 and Table 3 show how much of the time the rebalancing process spends idling because of an empty problem queue. For 16 or more processes, the rebalancing process always works to capacity. For 8 processes or fewer, the chromatic tree needs the least rebalancing work. The case of two concurrent processes depicts the differences between the algorithms most clearly. With a growing number of concurrent processes, the portion of the time the rebalancing process spends idling 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 Larsen's relaxed red-black tree.

Since Larsen's algorithm does not produce considerably more rebalancing requests than the other algorithms, we do not expect such a low idle time. In order to find the reasons for this, we have measured how often the rebalancing process fails to apply a rebalancing operation. The rate of failure to apply a red-balancing transformation is always insignificant. The rate of failure to apply a weight-balancing transformation shows considerable differences between the algorithms and gives reasons for the relatively low idle time of the rebalancing process in Larsen's algorithm.

In the chromatic tree, the rebalancing process never fails to handle an overweight conflict. In the basic relaxed balancing algorithm, the number of failures is almost always very low and is at most 2.7% of all overweight handling. Larsen's relaxed red-black tree differs crucially from the other two algorithms. Here, the rate of failure to handle an overweight conflict is up to 99% of all overweight handling. Furthermore, it depends only on the weight-temp operation. In all cases in which the tree is relatively well balanced, most of the time the rebalancing process tries to prepare weight-temp, fails because of an interfering overweight conflict in the vicinity, and appends the request again to the problem queue.

The high rate of failure is caused by the use of a FIFO problem queue. A FIFO queue supports the top-down handling of rebalancing requests, since requests are appended to the queue in the order in which they are created. This is very useful when resolving red-clusters in relaxed red-black trees, because red-red conflicts can be handled only in a top-down manner. On the other hand, weight-temp introduces a bottom-up handling of overweight-conflicts. Therefore, if a rebalancing process gets its work from a FIFO problem queue, it must use an additional strategy for the order in which overweight conflicts can be handled.

We have experimentally studied the effect of trying to handle the interfering overweight conflict next, if weight-temp cannot be applied. In the case of randomly chosen dictionary operations, this strategy leads to a satisfying decrease in the rate of failure.

Let us now compare the performance of the weight-balancing transformations. Since the weight-balancing operations of Larsen's algorithm are more grained than the ones of the chromatic tree or the basic relaxed balanced red-black tree, it could be expected that, on average, they need significantly less time to handle an overweight conflict. However, whenever the rate of failure during an experiment is only minimal, the chromatic tree and Larsen's relaxed red-black

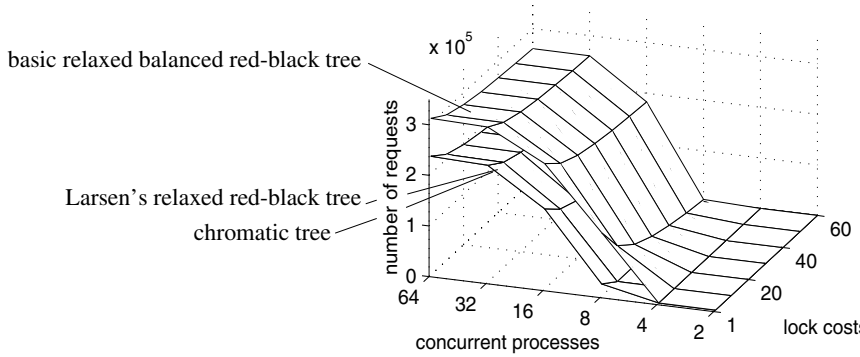


Fig. 5. Number of unsettled rebalancing requests.

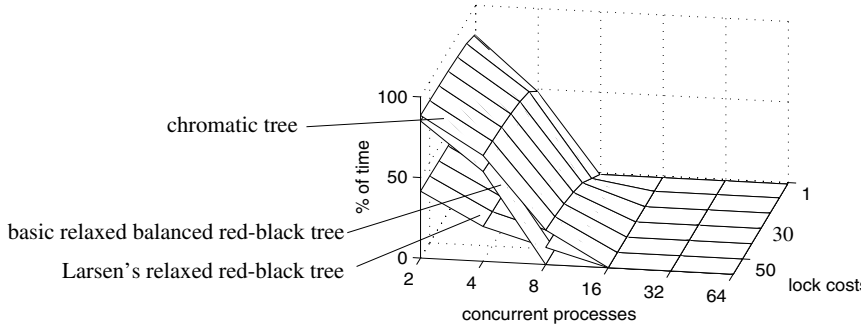


Fig. 6. Portion of the time the rebalancing process spends idling.

	number of processes											
	2	4	8	16	2	4	8	16	2	4	8	16
1	82.66	49.60	0.00	0.00	75.47	29.83	0.00	0.00	38.62	15.74	0.00	0.00
10	85.62	57.82	6.00	0.00	80.48	43.18	0.00	0.00	40.07	18.72	1.88	0.00
20	86.79	61.14	12.82	0.00	82.45	48.63	0.00	0.00	40.65	19.93	5.75	0.00
30	87.51	63.11	16.47	0.00	83.65	51.80	0.00	0.00	41.01	20.63	7.86	0.00
40	87.95	64.32	18.75	0.00	84.36	53.69	0.00	0.00	41.22	21.07	9.17	0.00
50	88.25	65.13	20.27	0.00	84.80	54.90	0.00	0.00	41.37	21.36	10.05	0.00
60	88.46	65.71	21.38	0.00	85.11	55.75	0.00	0.00	41.47	21.57	10.68	0.00
lock costs	*chromatic tree*				*basic relaxed balanced red-black tree*				*Larsen's relaxed red-black tree*			

Table 3. Portion of the time (in %) the rebalancing process spends idling.

tree perform in a broadly similar fashion. This is caused by the fact that the cases in which the handling of an overweight conflict differ in both algorithms occur very seldom. In less than 1.5% of the cases of overweight handling, one of the larger weight-balancing transformations apply in the chromatic tree. This corresponds exactly to the occurrence of weight-temp during the rebalancing of Larsen's relaxed red-black tree. Therefore, by the results of our experiments, we could not confirm that Larsen's relaxed red-black tree fits better in a concurrent environment.

4 Conclusions

We have experimentally studied the performance of three relaxed balancing algorithms for red-black trees, the chromatic tree [2], the algorithm that is obtained by applying the general method of how relaxing a strict balancing scheme to red-black trees [8], and Larsen's relaxed red-black tree [10], and we compared them with the performance of the strictly balanced red-black tree by Sarnak and Tarjan [16], combined with the locking scheme of Ellis [4]. We find 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 shows 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 relaxing strict balancing schemes [8] generates satisfactory relaxed balancing algorithms if this method is applied to other classes of search trees as well.

Regarding the quality of the rebalancing, there are differences between the three algorithms. We find that the chromatic tree always has the best performance. The rebalancing performance of the algorithm obtained by the general method is considerably lower than the performance of the chromatic tree, but it is still satisfactory. The rebalancing of Larsen's relaxed red-black tree requires the use of additional strategies for handling overweight-conflicts, since a FIFO problem queue and the need for the bottom-up handling of overweight conflicts leads to a considerably high rate of failure. Furthermore, the hoped for increase in the speed of the weight-balancing is insignificant since, during the experiments, one of the larger weight-balancing transformations applies to only less than 1.5% of all overweight handling.

Acknowledgments

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 the concurrent search tree algorithms. I would also like to thank Eljas Soisalon-Soininen and Thomas Ottmann, who have decisively contributed to the origin of this work.

References

1. 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*, pages 321–429. LNCS 1300, 1997.
2. J. Boyar, R. Fagerberg, and K. Larsen. Amortization results for chromatic search trees, with an application to priority queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997.
3. J. Boyar and K. Larsen. Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences*, 49:667–682, 1994.
4. C.S. Ellis. Concurrent search in AVL-trees. *IEEE Trans. on Computers*, C-29(29):811–817, 1980.
5. J. Gabarró, X. Messeguer, and D. Riu. Concurrent rebalancing on hyperred-black trees. In *Proceedings of the 17th Intern. Conference of the Chilean Computer Science Society*, pages 93–104. IEEE Computer Society Press, 1997.
6. 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.
7. S. Hanke. The performance of concurrent red-black tree algorithms. Technical Report 115, Institut für Informatik, Universität Freiburg, Germany, 1998.
8. S. Hanke, T. Ottmann, and E. Soisalon-Soininen. Relaxed balanced red-black trees. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, pages 193–204. LNCS 1203, 1997.
9. T. Johnson and D. Shasha. The performance of current B-tree algorithms. *ACM Trans. on Database Systems*, 18(1):51–101, March 1993.
10. K. Larsen. Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica*, 35(10):859–874, 1998.
11. L. Malmi. A new method for updating and rebalancing tree-type main memory dictionaries. *Nordic Journal of Computing*, 3:111–130, 1996.
12. L. Malmi. *On Updating and Balancing Relaxed Balanced Search Trees in Main Memory*. PhD thesis, Helsinki University of Technology, 1997.
13. L. Malmi and E. Soisalon-Soininen. Group updates for relaxed height-balanced trees. In *ACM Symposium on the Principles of Database Systems*, June 1999.
14. O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica 33*, pages 547–557, 1996.
15. K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylönen. Concurrency control in B-trees with batch updates. *Trans. on Knowledge and Data Engineering*, 8(6):975–983, 1996.
16. N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.
17. 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.
18. V. Srinivasan and M.J. Carey. Performance of B-tree concurrency control algorithms. *Proc. ACM Intern. Conf. on Management of Data*, pages 416–425, 1991.