CBTree: A Practical Concurrent Self-Adjusting Search Tree

by

Boris Korenfeld

A Thesis submitted for the degree

Master of Computer Science

Supervised by

Professor Yehuda Afek

School of Computer Science

Tel Aviv University

May 2012

CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my supervisor Prof. Yehuda Afek and Mr. Adam Morrison, for their guidance and support during my research. I would also like to thank Prof. Haim Kaplan and Prof. Robert E. Tarjan for their contribution to this work.

ABSTRACT

We present the CBTree, a new *counting-based* self-adjusting search tree that like *splay trees*, moves more frequently accessed nodes closer to the root. After m operations on n items, q of which access some item v, an operation on v traverses a path of length $\mathcal{O}(\log \frac{m}{q})$ while performing few if any rotations. In contrast to the traditional self-adjusting splay tree in which each accessed item is moved to the root through a sequence of tree rotations, CBTree performs rotations infrequently (an amortized subconstant $o(1)$ per operation if $m \gg n$), mostly at the bottom of the tree, therefore CBTree scales with the concurrency level. We adapt CBTree to a multicore setting and show experimentally that it improves performance compared to existing concurrent search trees on non-uniform access sequences derived from real workloads.

CBTree achieves the above by trading-off rotations for keeping history. Each node maintains a count of the number of accesses to it, and CBTree uses the counters to decide when to do a rotation. CBTree's decision rule is inspired by the analysis of splaying, which we draw upon to prove that CBTree achieves similar time bounds to the splay tree while performing only few if any tree rotations per operation.

We present another new self-adjusting algorithm LazyCBTree. LazyCBTree is near optimal sequentially, however it improves the performance on multi-core machines in practice. LazyCBTree, the *lazy counting-based tree*, like CBTree counts accesses to nodes and moves frequently accessed subtrees towards the root, obtaining optimal tree structure in practice including in a sequential setting. Unlike CBTree, LazyCBTree *lazily* makes at most one local tree rotation on each lookup, usually requiring no restructuring at all. LazyCBTree thus avoids creating a sequential bottleneck.

We evaluate CBTree and LazyCBTree on real non-uniform access patterns and show that they are significantly improve performance compared to existing self-adjusting trees as well as balanced (non-adjusting) trees.

1. INTRODUCTION

The shift towards multicore processors raises the importance of optimizing concurrent data structures for workloads that arise *in practice*. Such workloads are often *non-uniform*, with some *popular* objects accessed more frequently than others; this has been consistently observed in measurement studies, e.g., for web document references [2, 7, 14, 19] or media file references [9, 10, 24, 26]. Therefore, in this thesis we develop a concurrent data structure that completes operations on popular items faster than on ones accessed less frequently, leading to increased overall performance in practice.

We focus on the binary search tree (BST), a fundamental data structure for maintaining ordered sets. It supports successor and range queries in addition to the standard `insert`, `lookup` and `delete` operations. The basic binary tree terminology and the description of the system model presented in Section 2.

To the best of our knowledge, no existing concurrent BST is *self-adjusting*, adapting its structure to the access pattern to provide faster accesses to popular items. Most concurrent algorithms (e.g., [8, 15]) are based on sequential BSTs that restructure the tree to keep its height logarithmic in its size. The restructuring rules of these search trees do not prioritize popular items and therefore they do not provide optimal performance for skewed and changing usage patterns. Unlike these BSTs, known sequential self-adjusting trees do not lend themselves to an efficient concurrent implementation. A natural candidate would be Sleator and Tarjan's seminal *splay tree* [23], which moves each accessed node to the root using a sequence of rotations called *splaying*. The amortized access time of a splay tree for an item v which is accessed q_v times in a sequence of m operations is $\mathcal{O}(\log \frac{m}{q_v})$, matching, up to a constant factor, the information-theoretic optimum [18].

Unfortunately, splaying creates a major scalability problem in a concurrent setting. Every operation moves the accessed item to the root, turning the root into a hot spot, making the algorithm non-scalable. Additional self-adjusting trees as Treap [21] and Biased search trees [5] also lack an efficient concurrent implementation. We discuss the limitations these sequential self-adjusting BSTs as well as concurrent BSTs algorithms in Section 3. The bottom line is that no existing algorithm adjusts the tree to the access pattern *in practice* while still admitting a scalable concurrent implementation.

In Section 4, we present a *counting-based tree*, which we call CBTree for short, a self-adjusting BST that scales with the concurrency level, and has performance guarantees similar to the splay tree. CBTree maintains a *weight* for each subtree S which equals to the total number of accesses to items in S. CBTree operations use rotation in a way similar to splay trees, but rather than performing them at each node along the access path, it decides where to rotate based on the weights. CBTree does rotations to guarantee that the weights along the access path decrease geometrically, thus yielding a path of logarithmic length. Specifically, after m operations, q_v of which access item v, an operation on v takes $\mathcal{O}(\log \frac{m}{q_v})$ time.

CBTree's crucial performance property is that it performs only a *subconstant* ($o(1)$) amortized number of rotations per operation, so most CBTree operations perform few if any rotations. This allows the CBTree to scale with the concurrency level by avoiding the rotation-related synchronization bottlenecks that the splay tree experiences. Thus the performance gain by eliminating rotations using the counters outweighs losing the splay tree's feature of not storing book-keeping data in the nodes. This is similar to how one avoids false sharing in concurrent data structures by placing certain fields in a cache line of their own. CBTree replaces most of the rotations splaying does with counter updates, which are local and do not change the structure of the tree.

In Section 5 we present the concurrent implementation of the CBTree. To translate CBTree's theoretical properties into good performance in practice, we minimize the synchronization costs associated with the counters. First, we maintain the counters using plain reads and writes, without locking, accepting an occasional lost update due to a race condition; we show experimentally that CBTree is robust to inaccuracies due to data races on these counters.

Yet even plain counter updates are overhead compared to the read-only traversals of traditional balanced BSTs. Moreover, we observe that updates of concurrent counter can limit scalability on a multicore architecture where writes occur in a core's private cache (as in Intel's Xeon E7) instead of in a shared lower-level cache (as in Sun's UltraSPARC T2). We thus develop a *single adjuster* optimization in which an adjuster thread performs the self-adjusting as dictated by its own accesses and other threads do not update counters. When all the threads' accesses come from the same workload (same distribution), the adjuster's operations are representative of all threads, so the tree structure is good and performance improves as most threads perform read-only traversals without causing serialization on counter cache lines. If the threads have different access patterns, the resulting structure will probably be poor no matter what, since different threads have different popular items.

In addition, we describe how to exponentially decay CBTree's counters over time so that it responds faster to a change in the access pattern.

The calculations of the potential in all nodes on the path from root to the required node are still expensive in single-thread as well as in multi-thread implementations. To handle this problem we present another new self-adjusting algorithm LazyCBTree *lazy counting-based tree*. LazyCBTree is near optimal sequentially, however it improves the performance in practice. Unlike CBTree, LazyCBTree *lazily* makes at most one local tree rotation on each lookup, usually requiring no restructuring at all. Additionally LazyCBTree uses simpler calculation to decide whether to perform the rotations. These differences allow LazyCBTree to perform more operations per second than CBTree, though they both create trees with similar average path length. The sequential and concurrent LazyCBTree algorithm and its analysis presented in Section 6.

We emphasize that as any other binary search tree CBTree and LazyCBTree restructure themselves by rotations. Their advantage is in a clever way to decide *when* to rotate so that rotations are infrequent. It follows that one can implement CBTree or LazyCBTree easily on top of any concurrent code for doing rotations atomically. In our implementation we used Bronson et al.'s optimistic BST concurrency control technique [8]. In Section 7 we present the code of CBTree and LazyCBTree algorithms. We start from explaining Bronson et al.'s synchronization techniques. We continue by presenting CBTree restructuring code and implementations of lookup, insert and delete functions. Then we present the code that performs counters exponential decay. Finally, we show the LazyCBTree restructuring code which is the main difference from CBTree code.

In Section 8, we compare our CBTree and LazyCBTree implementations against other sequential and concurrent BSTs (also implemented using Bronson et al.'s technique) using real workloads. We show that CBTree and LazyCBTree provide short access paths and excellent throughput.

Finally, some directions for future work are suggested in Section 9.

2. MODEL AND BSTS

Binary search trees A *binary search tree* (BST) is a rooted tree in which each node v has at most two children, a left child, $v.left$, a right child, $v.right$, and an item, $v.key$, from a totally ordered domain. The *left subtree* of node v is the subtree rooted at $v.left$; the *right subtree* is defined symmetrically. The left subtree of v holds items smaller than $v.key$ and the right subtree holds items larger than $v.key$. BSTs support standard set `insert`, `remove` and `lookup` operations as well as range queries and ordered iteration. A set operation on an item first searches for the item, starting from the root and proceedings left or right at each node v if the searched item is smaller or larger than $v.key$, respectively, until finding the searched item or reaching an empty subtree (denote `null`) indicating the item is not in the tree. In such a case the item can be inserted at the point where the search stopped.

System model In the sequential domain, we use a standard system model where both visiting a BST node and performing a tree rotation have unit cost. When considering concurrency, we use a standard shared memory system model with a set of sequential asynchronous *threads* that communicate via shared memory.

3. RELATED WORK

Treaps A treap [21] is a BST in which each node has a priority in addition to an item, and every priority in a node's subtree does not exceed its own; this is called the *heap property*. Whenever a node v is accessed, a random number r is generated and it replaces v's priority if r is greater than it. If necessary, v is then rotated up in the tree until the heap property is restored. Treaps provide *probabilistic* bounds similar to those of the CBTree: after m accesses, q of which are to item v, v will be at expected depth $\mathcal{O}(\log \frac{m}{q})$ in the treap and accessing v will incur an expected $\mathcal{O}(1)$ rotations [21]. The treap's rotations are driven by local decisions, avoiding the synchronization bottlenecks caused by the splay tree. Treaps are therefore suitable for a concurrent implementation, and indeed our experiments show that treaps scale with the concurrency level. However, we find that in practice nodes' depth in the treap vary greatly from the expected bound. Consequentially, the CBTree (and splay tree) obtain better path length than the treaps (Section 8).

Biased search trees Several works in the sequential domain consider *biased* search trees where an item is *a priori* associated with a weight representing its access frequency. Bent, Sleator and Tarjan's discuss these variants extensively [5]. Biased trees allow the item weight to be changed using a `reweight` operation, and can therefore be adapted to our dynamic setting by following each access with a `reweight` to increment the accessed item's weight. However, most biased tree algorithms do not easily admit an efficient implementation. In the biased trees of [5, 13], for example, every operation is implemented using global tree splits and joins, and items are only stored in the leaves. CBTree's rotations are the same as those in Baer's weight-balanced tree [3], however CBTree uses different rules to decide when to apply rotations. CBTree's analysis is based on the analysis of splaying whereas Baer did not provide a theoretical analysis. Baer also did not consider concurrency.

Concurrent ordered map algorithms Most existing concurrent BSTs are balanced and rely on locking, e.g. [8, 15]. Ellen et al. proposed a nonblocking concurrent BST [12]. Their tree is not balanced, and their focus was obtaining a lock-free BST. Crain et al.'s recent transaction-friendly BST [11] is a balanced tree where a dedicated thread continuously scans the tree looking for balancing rule violations that it then corrects. Unlike CBTree's adjuster, this thread does not perform other BST operations. Furthermore, the motivation is to reduce rotation-related aborts in the context of transactional memory. Ordered sets and maps can also be implemented using skip lists [20], which are also not self-adjusting.

4. CBTree AND ITS ANALYSIS

4.1 Splaying analysis background

CBTree's design draws from the analysis of semi-splaying, a variant of splaying [23]. To (bottom-up) *semi-splay* an item v known to be in the tree, an operation first locates v in the tree in the usual way. It then ascends from v towards the root, restructuring the tree using rotations as it goes. At each *step*, the operation looks up two nodes along the path to the root and decides which rotation(s) to perform according to the structure of the path, as depicted in Figure 4.1. Following the rotation(s) it continues from the node which replaces the current node's grandparent in the tree (in Figure 4.1, this is node y after a single rotation and node x after a double rotation). If only one node remains on the path to the root then the final edge on the path is rotated.

To analyze the amortized performance of splaying and semi-splaying Sleator and Tarjan use the potential method [25]. The potential of a splay tree is defined based on

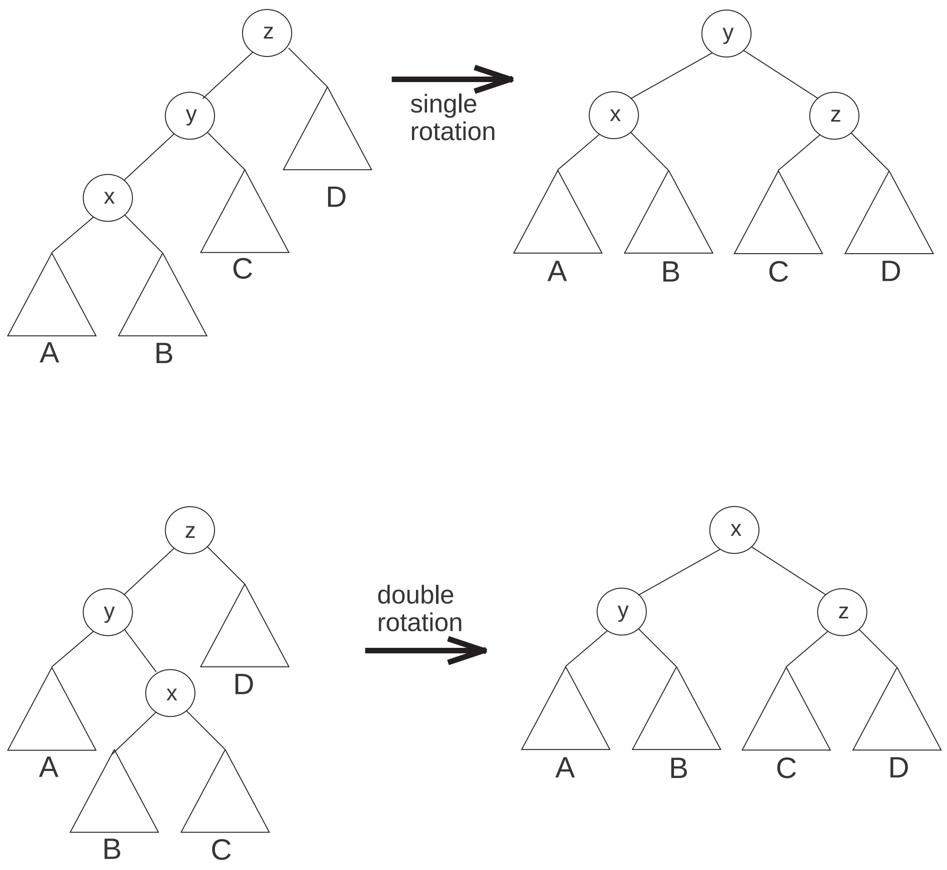


Fig. 4.1: Semi-splaying restructuring: current node is x, and the next two nodes on the path to the root are y and z. The case when y is a right child is symmetric.

an arbitrary but fixed positive weight which is assigned to each item.[1] The splay tree algorithm does not maintain these weights, they are defined for the analysis only.

Let $c(v)$ be the weight assigned to node v, and let $W(v)$ be the total weight of the nodes currently in the subtree rooted at v. Let $r(v) = \lg W(v)$ be the *rank* of v.[2] The *potential* of a splay tree is $\Phi = \sum r(v)$ over all nodes v in the tree. Sleator and Tarjan's analysis of semi-splaying relies on the following bound for the potential change caused by a rotation [23]:

Lemma 1. *Let Φ, and Φ' be the potentials of a splay tree before and after a semi-splay step at node x, respectively. Let z be the grandparent of x before the semi-splay step (as in Figure 4.1), and let $\Delta\Phi = \Phi' - \Phi$. Then*

$$2 + \Delta\Phi \leq 2(r(z) - r(x))$$

where $r(x)$ and $r(z)$ are the ranks of x and z in the tree before the step, respectively.

The analysis uses this lemma to show that the amortized time of semi-splaying node v in a tree rooted at *root* is $\mathcal{O}(r(root) - r(v)) = \mathcal{O}(\lg(W(root)/W(v)))$. This holds for any assignment of node weights; different selections of weights yield different bounds, such as bounds that depend on access frequencies of the items or bounds that capture other patterns in the access sequence. Here we focus on using a node's access frequency as its weight, i.e., given a sequence of m tree operations let $c(v) = q_v$ be the number of operations on v in the sequence. Using Lemma 1 with this weight assignment Sleator and Tarjan proved the following [23]:

Theorem 1 (Static Optimality of Semi-Splaying)**.** *The total time for a sequence of m semi-splay operations on a tree with n items $v_1, \ldots, v_n$, where every item is accessed at least once, is*

$$\mathcal{O}\left(m + \sum_{i=1}^{n} q_{v_i} \lg \frac{m}{q_{v_i}}\right),$$

where q_v is the number of times v is accessed in the sequence.

Hereafter we say that $\mathcal{O}(\lg(m/q_v))$ is the *ideal access time* of v.

4.2 Sequential CBTree

A CBTree is a binary search tree where each node contains an item; we use the terms *item* and *node* interchangeably. We maintain in any node v a *weight* $W(v)$ which counts the total number of operations on v and its descendants. We can compute $c(v)$, the number of operations on v, from the weight of v and its children by $c(v) = W(v) - (W(v.left) + W(v.right))$, using a weight of 0 for null children.

A CBTree operation performs steps similar to semi-splaying, however it does them in a top-down manner and it decides whether to perform rotations based on the weights of the relevant nodes.

Lookup: To lookup an item v we descend from the root to v, possibly making rotations along the way down. We maintain a current node z along the path to v and look two nodes ahead along the path to decide whether to perform a single or a double rotation. Assume that the next node y along the path is the left child of z (the case when it is a right child is symmetric). If the child of y along the path is also a left child we may decide to perform a single rotation as in the top part of Figure 4.1. If the child of y along the path is a right child we may perform a double rotation as in

[1] The splay algorithm never changes the node containing an item, so we can also think of this as the weight of the node containing the item.

[2] We use lg to denote $\log_2$.

the bottom part of Figure 4.1. From here on, unless we need to distinguish between the cases, we refer to a single or a double rotation simply as a *rotation*.

We perform a rotation only if it would decrease the potential of the tree by at least an additive factor $\epsilon \in (0, 2)$, i.e., if $\Delta\Phi < -\epsilon$. After performing a rotation at z, the CBTree operation changes the current node to be the node that replaces z in the tree, i.e., node y after a single rotation or node x after a double rotation. If we do not perform a rotation, the current node *skips ahead* from z to x without restructuring the tree. (Naturally, if the search cannot continue past y the search moves to y instead.) A search whose traversal ends finding the desired item v increments $W(v)$ and then proceeds in a bottom-up manner to increment the weights of all the nodes on the path from v to the root.

To summarize, during a search the parent of the current node stays the same and we keep reducing the potential by at least ϵ using rotations until it is no longer possible. We then advance the current node two steps ahead to its grandchild.

We note that Sleator and Tarjan describe a top-down semi-splaying variant [23]. This variant maintains four trees, moving nodes from the original tree to these trees as it searches for v, and reassembling the outcome tree from them at the end. In contrast, the CBTree algorithm naturally works top-down.

Insert: An `insert` of v searches for v while restructuring the tree, as in a lookup. If v is not found, we replace the null pointer where the search terminates with a pointer to a new node containing v with $W(v) = 1$, then increment the weights along the path from v to the root. If v is found we increment $W(v)$ and the weights along the path and consider the `insert` failed.[3]

Remove: We remove an item by marking its node as deleted and deferring actually unlinking it from the tree until it becomes a leaf or has a single child. When an item is finally unlinked we do not remove its weight from the tree, in effect remembering the item's history in case it is inserted again. We can therefore no longer compute an item's individual weight, $c(v)$, from the weight of its subtree, $W(v)$, requiring us to maintain them separately. We discuss the details in Section 7.5, but avoid burdening the presentation here and focus on inserts and lookups.

Computing potential differences: An operation needs to compute $\Delta\Phi$ to decide whether to perform a rotation. It can do this using only the counters of the nodes involved in the rotation, as follows. Consider first the single rotation case of Figure 4.1 (the other cases are symmetric). Only nodes whose subtrees change contribute to the potential change, so $\Delta\Phi = r'(z) + r'(y) - r(z) - r(y)$, where $r'(v)$ denotes the rank of v after the rotation. Because the overall weight of the subtree rooted at z does not change, $r'(y) = r(z)$ and thereby $\Delta\Phi = r'(z) - r(y)$. We can express $r'(z)$ using the nodes and their weights in the tree before the rotation to obtain

$$\begin{aligned}\Delta\Phi = &\lg(c(z) + W(y.right) + \\ &W(z.right)) - \lg W(y).\end{aligned} \tag{4.1}$$

For a double rotation, an analogous derivation yields

$$\begin{aligned}\Delta\Phi = &\lg\left(c(z) + W(x.right) + W(z.right)\right) + \\ &\lg\left(c(y) + W(y.left) + W(x.left)\right) - \\ &\lg W(y) - \lg W(x).\end{aligned} \tag{4.2}$$

When we do a rotation, we also update the weights of the node involved in the rotation.

[3] If we think of items as keys in a dictionary, then `insert` can take both item v and an associated value w, and a failed `insert` updates v's value to w.

4.3 Analysis

In this section we consider only *successful* lookups, that is, lookups of items that are in the tree. We prove the following bound on the time of a sequence of operations on a CBTree.

Theorem 2. *Consider a sequence of m `insert` and `lookup` operations on n distinct items, starting with an initially empty CBTree. Then the total time it takes to perform the sequence is*

$$\mathcal{O}\left(n\ln\frac{m}{n}+\sum_{i=1}^{m}\lg\frac{i}{c_i(v_i)}\right),$$

where v_i is the item operation i accesses, and $c_i(v_i)$ is the number of operations $j \leq i$ which apply to v_i.

An insert or lookup operation on item v does two kinds of steps: (1) rotations, and (2) traversals of edges in between rotations. The edges traversed in between rotations are exactly the ones on the path to v at the end of the operation. Our proof of Theorem 2 accounts separately for the total time spent traversing edges in between rotations and the total time spent doing rotations.

We prove that an operation on node v, applied to a CBTree with weights $W(u)$ for each node u, traverses $\mathcal{O}(\log(W/c(v))$ edges in between rotations, where $W = W(root)$ is the weight of the entire tree and $c(v)$ is the individual weight of v (recall $c(v) = W(v) - (W(v.left) + W(v.right))$. Applying this to our sequence of operations, we see that operation i traverses $\mathcal{O}(\log(i/c_i(v))$ edges in between rotations, where $c_i(v)$ is the number of accesses to v in the prefix of length i of the sequence. Having established this, the amortized bound in Theorem 2 follows by showing that the total number of rotations in all m operations in the sequence is $\mathcal{O}(n \ln \frac{m}{n})$.

Lemma 2 (Ideal access path)**.** *Consider a CBTree with weights $W(u)$ for each node u. The length of the path traversed by an operation on item v is $\mathcal{O}(\lg(W/c(v)))$, where $W = W(root)$ and $c(v)$ is the individual weight of v, at the time the operation is performed.*

Proof. The path to v following the operation consists of d pairs of edges (z, y) and (y, x) that the operation skipped between rotation, and possibly a single final edge if the operation could look ahead only through a single edge at its final step. For each such pair (z, y) and (y, x), let $\Delta\Phi$ be the potential decrease we would have got by performing a rotation at z. Since we have not performed a rotation, $\Delta\Phi > -\epsilon$. By Lemma 1 we obtain that

$$2(r(z) - r(x)) \geq 2 + \Delta\Phi > 2 - \epsilon. \tag{4.3}$$

Define $\delta = 1 - \epsilon/2$. Then Equation (4.3) says that $r(z) - r(x) > \delta$ for each such pair of edges (z, y) and (y, x) on the final path. Summing over the d pairs on the path we get that $r(root) - r(v) > d\delta$ and so

$$\begin{aligned} d < \frac{r(root) - r(v)}{\delta} &= \mathcal{O}\left(\lg\frac{W(root)}{W(v)}\right) \\ &= \mathcal{O}\left(\lg\frac{W}{c(v)}\right). \end{aligned}$$

Since the length of path to v is at most $2d + 1$, the lemma follows. □

We now bound the number of rotations.

Lemma 3. *In a sequence of m insert and lookup operations starting with an empty CBTree, we perform $O(n \ln \frac{m}{n})$ rotations, where n is the number of insertions.*

Proof. Since each rotation decreases the potential by at least ϵ, and the potential decreases only by rotations, the number of rotations is bounded by the sum of potential increases throughout the sequence.

The potential increases when we increment the weight of v following an operation at v. Let P be the path from the root to v when we reach v, after performing all rotations, then by incrementing $c(v)$ we increase the potential by

$$\begin{aligned}\sum_{u \in P} \lg(W(u)+1) - \lg(W(u)) &= \sum_{u \in P} \lg \frac{W(u)+1}{W(u)} \\ &= \sum_{u \in P} \lg(1 + \frac{1}{W(u)}) \leq \sum_{u \in P} \frac{1}{W(u)}.\end{aligned}$$

The proof of Lemma 2 implies that P (except possibly for its last edge) consists of d pairs of edges (z, y), and (y, x), such that $r(z) - r(x) > \delta$ or equivalently $W(z)/W(x) > 2^{\delta}$. Let P' be the subset of the nodes $u \in P$ which are either z or x in a pair (z, y), and (y, x). It follows that $1/W(u)$ for $u \in P'$ is a geometric series. Let $\Gamma = \sum_{u \in P'} \frac{1}{W(u)}$ denote the sum of this series. We have that $\Gamma = \mathcal{O}(1/W(v)) = \mathcal{O}(1/c(v))$ since the largest term in the series is either $1/W(v)$ or $1/W(p(v))$, where $p(v)$ is v's parent.

Consider a node $u \in P \setminus P'$. Then u is either a node y in a pair $(z, y), (y, x)$ or the target node v and in either case $p(u) \in P'$. So

$$\sum_{u \in P} \frac{1}{W(u)} \leq 2\Gamma = \mathcal{O}\left(\frac{1}{c(v)}\right) .$$

Consider now all operations on a particular node v over the entire sequence. After the k-th time v is accessed, its individual weight is k. Thus, the potential increase due to all operations that access v is

$$\sum_{i \,:\, \text{operation } i \text{ accesses } v} \mathcal{O}\left(\frac{1}{c_i(v)}\right) = \mathcal{O}(\ln c(v)),$$

where $c_i(v)$ is v's individual weight after operation i.

By summing this over the distinct items accessed in the sequence, we have that the overall number of rotations is bounded by

$$\sum_{v} \mathcal{O}(\ln c(v)) \leq \mathcal{O}\left(n \ln \frac{m}{n}\right)$$

since $\sum_{i=1}^{n} \ln x_i$ where $\sum_{i=1}^{n} x_i \leq m$ is maximized when all x_i's are equal. □

5. CONCURRENT CBTree

We demonstrate the CBTree using Bronson et al.'s optimistic BST concurrency control technique [8] to handle synchronization of generic BST operations, such as rotations and node link/unlinks. To be self contained, we summarize this technique in Section 5.1. Section 5.2 then describes how we incorporate CBTree into it. Section 5.3 describes our *single-adjuster* optimization. Pseudo-code is relegated to Section 7.

5.1 Optimistic concurrent BSTs

Bronson et al. use techniques inspired by software transactional memory to implement traversal through the tree without relying on read-write locks, using *hand-over-hand optimistic validation* instead. This is similar to hand-over-hand locking [4] except that instead of overlapping lock-protected sections, we overlap atomic blocks which traverse node links. Atomicity within a block is established using versioning. Each node holds a version number with reserved `changing` bits to indicate that the node is being modified. A navigating reader (1) reads the version number and waits for the `changing` bits to clear, (2) reads the desired fields from the node, (3) rereads the version. If the version has not changed, the reads are atomic. Section 7.1 describes the implementation of these ideas in detail.

5.2 Concurrent CBTree walk-through

We represent a node v's weight counter $W(v)$, which counts the total number of operations on v and its descendants, with three counters: $selfCnt$, counting the total number of operations on v, $rightCnt$ for the total number of operations on items in v's right subtree, and $leftCnt$ which is an analogous counter for the left subtree.

Traversal: Hand-over-hand validation works by chaining short atomic sections using recursive calls. Each section traverses through a single node u after validating that both the *inbound* link, from u's parent to u, and the *outbound* link, from u to the next node on the path, were valid together at some point in time (see Section 7.1). If the validation fails, the recursive call returns so that the previous node can revalidate itself before trying again. Eventually the recursion unfolds bottom-up back to a consistent state from which the traversal continues.

When traversing through a node (i.e., at each recursive call) the traversal may perform a rotation. We describe the implementation of rotations and of the rotation decision rule below. For now, observe that performing a rotation invalidates both the inbound and outbound links of the current node. Therefore, after performing a rotation the traversal returns to the previous recursion step so that the caller revalidates itself. Using Figure 4.1's terminology, after performing a rotation at z the recursion returns to z's parent (previous block in the recursion chain) and therefore continues from the node v that replaces z in the tree. (See Section 7.2 for pseudo-code and a detailed explanation.)

In doing this, we establish that a traversed link is always verified by the hand-over-hand validation, as in the original optimistic validation protocol. The linearizability [16] of CBTree therefore follows from the linearizability of Bronson et al.'s algorithm.

Rotations: To do a rotation, CBTree first acquires locks on the nodes whose links are about to change in parent-to-child order. Using Figure 4.1's terminology these are z's parent, z, y, and for a double rotation also x. Having acquired the locks, CBTree validates that the relationship between the nodes did not change and then performs the rotation which is done exactly as in Bronson et al.'s algorithm, changing node version number as required and so on. After the rotation the counters are updated to reflect the number of accesses to the new subtrees. (Section 7.3 describes the details and code.)

Counter maintenance: Maintaining consistent counters requires synchronizing with concurrent traversals and rotations. Traversals can synchronize by atomically incrementing the counters using `compare-and-swap`, but this does not solve the coordination problem between rotations and traversals. While traversals could synchronize with rotations by acquiring each traversed node's lock, that would ruin the algorithm's scalability by turning the top of the tree into a hot spot.

Thus, keeping the counters consistent requires an *additional* mechanism to synchronize between rotations and counter-updating in traversals. But any such synchronization would be pure overhead in the common case, because CBTree's analysis and our empirical evidence show that rotations are rare. We therefore choose to do without consistent counters and access counters using plain reads and writes, observing that wrong counter values will not violate the algorithm's correctness, only possibly its performance.

A traversal increments the appropriate counters as the recursion unfolds after a successful completion of the operation (i.e., not during the intermediate retries that may occur). Consequentially, a traversal that visits the same node twice will increment the same counter twice. This is acceptable because it can happen only due to a rare concurrent rotation: the scenario is for a thread T to be at node x (bottom part of Figure 4.1) when a concurrent double rotation occurs at node z, moving x above z but not invalidating T's traversal (see Section 7).

Similarly, traversals and rotations may overwrite each other's counter updates. (But two rotations cannot, as they are synchronized by locks.) Again, we accept this and our experiments show that such races – if they occur – do not keep CBTree from obtaining short access paths for frequent items.

Insert (Section 7.4): Insertion is implemented with a recursive traversal which updates counters as it unfolds, as explained above. The traversal terminates by adding a new item or updating the current item's value.

Remove: Our `remove` implementation resembles Bronson et al.'s approach [8], marking an node as remove and opportunistically unlinking it from the tree when it has less than two children. Section 7.5 describes this in detail.

Speeding up adaptation to access pattern change: After a change in the access pattern, i.e., when a popular item becomes unpopular, frequent nodes from the new distribution may take a lot of time until their counters are high enough to beat nodes that lost their popularity. To avoid this problem we add an exponential decay function to the counters, based on an *external clock* that is updated by an auxiliary thread or by the hardware. We detail this technique in Section 7.6. We note here that the decaying is an infrequent event performed by threads as they traverse the tree. Therefore decaying updates can also be lost due to races, which we again accept since the decaying is an optimization that has no impact on correctness.

5.3 Single adjuster

Even relaxed counter maintenance can still degrade scalability on multicore architectures such as Intel's Xeon E7, where a memory update occurs in a core's private cache, after the core acquires exclusive ownership of the cache line. See section 8.1 for details.

To bypass this Intel architectural limitation, we propose an optimization in which only a single *adjuster* thread performs counter updates during its `lookup`s. The remaining threads perform read-only `lookup`s. Thus, only the adjuster thread requires exclusive ownership of counter cache lines; other `lookup`s request shared ownership, allowing their cache misses to be handled in parallel. Similarly, when the adjuster writes to a counter, the hardware sends the invalidation requests in parallel. Synchronization can be further reduced by periodically switching the adjuster to read-only mode.

While the single-adjuster optimization causes counter updates during `lookup` to represent only the access distribution of the adjuster thread, if all threads' accesses come from the same workload (same distribution) they will all benefit from the adjuster's restructuring.

6. LazyCBTree AND ITS ANALYSIS

In LazyCBTree each node x has three counters: `selfCnt`, counting the total number of accesses to x (number of `insert(`x`)` and `lookup(`x`)`), `rightCnt` for the total number of accesses to items in x's right subtree, and `leftCnt` which is an analogous counter for the left subtree. A successful `insert(`x`)` or `lookup(`x`)` increments x.`selfCnt` and then proceeds in a bottom-up manner to increment the `rightCnt`/`leftCnt` counter of all the nodes on the path from x to the root, depending on whether x is in their right or left subtree. Maintaining counters obviates one of the original advantages of splay trees, which do not store balancing-related information at the nodes. However, we believe that in a modern environment the memory overhead of counters is negligible compared to the scalability gains they enable.

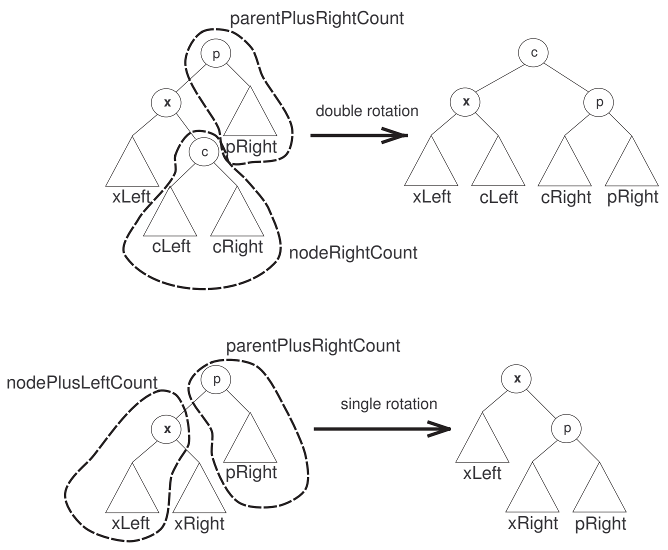


Fig. 6.1: LazyCBTree restructuring.
Double rotation: $x.rightCnt \geq p.selfCnt + p.rightCnt$.
Single rotation: $x.selfCnt + x.leftCnt > p.selfCnt + p.rightCnt$.

In the following, let $cnt(u)$ be the total number of accesses to node u's subtree, i.e., $cnt(u) = u.selfCnt + u.rightCnt + u.leftCnt$. For a LazyCBTree T, we define $cnt(T)$ as $cnt(r)$, where r is T's root.

6.1 *Self-adjusting rule*

We now describe LazyCBTree's self-adjusting restructuring rule. The idea is to estimate, based on past access frequencies, whether the total cost of accesses to x's neighborhood in the tree would have been reduced had a certain rotation been performed. If so, this rotation is performed under the assumption that past behavior is a good predictor for the future usage pattern. Below we detail the restructuring rule assuming that x is a left child; the case of a right child is symmetric. The restructuring rule is also depicted in Figure 6.1.

```
for (i=0; i < m; i += 3) {
    insert(i);
    insert(i+2);
    insert(i+1);
    lookup(i+1);
}
```

Listing 6.1: Example of how one restructuring step on `insert()` of new node leads to $\Omega(n)$ amortized complexity.

1. **Double rotation:** If the total number of accesses to x's right subtree is greater than or equal to the total number of accesses to x's parent and its right subtree.
2. **Single rotation:** If a double rotation is not performed, and if the total number of accesses to x and its left subtree is greater than the total number of accesses to x's parent and its right subtree.

Observe that when LazyCBTree restructures the tree, the average path length in the tree decreases, as the length of the path from the root to the heavier subtrees decreases.

6.2 Applying the rule

A `lookup(`x`)` or an `insert(`x`)` that finds x in the tree performs a single restructuring step at x, if needed, and then proceeds to update the counters along the path back to the root. In contrast, an `insert` that successfully adds a new node tries to restructure around each node along the path from the new node to the root as it walks back and updates the counters. To see why this is required, observe that if we perform only a single restructuring step when inserting a new node then the algorithm will have poor time complexity, even in an amortized sense. Listing 6.1 shows an access sequence that demonstrates this by creating a path of length n using $m = \frac{4}{3}n$ operations. Since each operation traverses the entire path, the sequence's total cost is $\Omega(n^2)$, i.e., the amortized complexity is $\Omega(n)$. See Section 7.7 for pseudocode.

6.3 Restructuring rule motivation:

Here we show that LazyCBTree essentially approximates an order optimum BST for a given access sequence. To do this we define a balance property which, when it holds for every node in the tree, results in an optimum BST.

Definition 1. *A node x is* frequency balanced *if $x.leftCnt \leq 2(x.selfCnt + x.rightCnt)$ and $x.rightCnt \leq 2(x.selfCnt + x.leftCnt)$.*

Theorem 3. *Let T be a LazyCBTree whose nodes are frequency balanced. Then each node x is in depth $\mathcal{O}(\log \frac{cnt(T)}{x.selfCnt})$.*

Proof. Let r be T's root, and let L be r's left child (a symmetric argument applies to the right child). By frequency balance, $cnt(L) \leq 2(r.selfCnt + r.rightCnt)$. Thus, $cnt(L) \leq 2(cnt(T) - cnt(L))$ implying that $cnt(L) \leq \frac{2}{3}cnt(T)$. It follows inductively that for a node x at depth d in T, $cnt(x) \leq (\frac{2}{3})^d cnt(T)$. Since $x.selfCnt \leq cnt(x)$, we have $d = \mathcal{O}(\log \frac{cnt(T)}{x.selfCnt})$. □

Since the cost of a LazyCBTree operation on a node is proportional to the node's depth, it follows that a frequency balanced LazyCBTree is optimum. It remains only to show that LazyCBTree restructuring rule implies frequency balance:

Lemma 4. *If both children of x do not require restructuring, then x is frequency balanced.*

Proof. Let L and R be x's left and right children. Suppose $cnt(R) \leq cnt(L)$ (the other case is symmetric). Then $x.rightCnt = cnt(R) \leq cnt(L) = x.leftCnt \leq 2(x.selfCnt + x.leftCnt)$. Because L does not require restructuring, we have (1) $L.selfCnt + L.leftCnt \leq x.selfCnt + x.rightCnt$ and (2) $L.rightCnt < x.selfCnt + x.rightCnt$. Summing these shows that $x.leftCnt = cnt(L) \leq 2(x.selfCnt + x.rightCnt)$. □

6.4 LazyCBTree is frequency balanced in practice:

LazyCBTree's restructuring does not guarantee that frequency balance will hold everywhere, for two reasons. First, a `lookup` may cause frequency balance to be violated above the accessed node, and the single restructuring step performed will not fix that. Second, a restructuring step may cause frequency balance to be violated *below* the node. Consider, for example, the single rotation step in Figure 6.1 where *xRight* is empty, but x is frequency balanced because $x.selfCnt$ compensates for this. Yet after the rotation, *xRight* becomes a subtree of p where this may not hold. We accept this type of approximation because *in practice, it produces a good tree structure*, as we now show. For each of our access sequences, we counted the numbers of two types of *violating nodes* in the tree after each operation: (1) a node where LazyCBTree would have performed a rotation had that node been looked up, and (2) nodes that are not frequency balanced. (Per Lemma 4, the number of type 1 violations is an upper bound on the number of type 2 violations.) In all cases the fraction of violating nodes quickly converged to a stable value. We present (in Table 6.1) the numbers at the end of the execution. As shown, there are very few violations. In particular, the fraction of nodes not frequency balanced is negligible. The table also shows LazyCBTree's average path length and throughput compared to the splay tree. LazyCBTree's path length is better than the splay tree's on **books**, comparable on **isp** and worse on **youtube**. However, each splay tree operation involves rotations along the path to the node, making the operation heavier than the corresponding LazyCBTree operation. In addition, the splay tree's performance does not scale in a concurrent setting, as shown in Section 8, whereas LazyCBTree's reliance only on localized restructuring admits an efficient concurrent implementation.

	Splay tree		LazyCBTree		LazyCBTree violating nodes	
	Path length	Ops/millisec	Path length	Ops/millisec	Type 1	Type 2
books	10.66	2256.37	8.10	3285.95	4.8%	0.01%
isp	9.36	3606.81	10.89	3923.70	4.3%	0.138%
youtube	8.83	5067.66	11.66	3831.68	2.1%	0.005%

Tab. 6.1: Empirical evidence that LazyCBTree's lazy restructuring leads to good tree structure in practice. Results from sequential execution on an Intel Nehalem.

7. ALGORITHM CODE

7.1 Optimistic concurrent BSTs

Listing 7.1 shows a pseudo code example for the `lookup()` operation. The search is implemented recursively; the `tryLookup` method implements a single recursion step, which tries to traverse one node by validating that both the link to the node from its parent and the link from the node to the next node on the path existed together in the tree at some point in time. If this validation fails `tryLookup()` returns a special `RETRY` value to its caller, which will then either revalidate its node and try to proceed again or fall back to its caller.

The version protecting the link between a parent and child is stored in the child. The `tryLookup()` method receives the version `vers` under which the pointer to `node` was read by the previous recursion step. Its purpose is to navigate in direction `dir` from node, so it reads the appropriate child pointer. If there is no child, it validates the node's version number and terminates (Lines 3-7). Otherwise, the next step in the navigation has been determined (Line 9). If the child is changing, the operation is retried (Lines 13-14). Otherwise, the child's version (protecting the link from `node` to `child`) is read and verified to be consistent by checking that `node` still points to `child` 15-16. Then the validity of the link to `node` is checked (Lines 17-18). If successful, these combined checks establish that both links existed in the tree together at some point in time and the navigation continues to the child.

The implementation distinguishes between rotations that move a node up the tree (*grows*) and down the tree (*shrinks*) by encoding the type in the version number. A reader encountering a grow is not in danger of missing its target, since a grow increases the range of items in the node's subtree, and the reader can therefore proceed. We omit the details of this from Listing 7.1.

Following Bronson et al. we assume a runtime system which automatically reclaims *garbage* memory that is not accessible to any thread, as is the case in modern environments like Java and C#. The validation technique relies on this assumption as it may read from a node after it has been unlinked from the tree (Listing 7.1, Line 16).

```
Object tryLookup(key, node, dir, vers) {
  while (true) {
    Node child = node.child(dir);
    if (child == null) {
      if (node.version != vers)
        return RETRY;
      return null;
    }
    nextdir = compare(key, child.key)
    if (nextdir == EQUAL)
      return child.value;
    long nextvers = child.version;
    if (nextvers is changing)
      waitUntilNotChanging(child);
    else if (child == node.child(dir)
             and child not unlinked) {
      if (node.version != vers)
        return RETRY;
      Object p = tryLookup(key, child, nextdir, nextvers);
      if (p != RETRY)
        return p;
} } }
```

Listing 7.1: BST navigation using hand-over-hand optimistic validation

7.2 CBTree lookup

Listing 7.2 shows how CBTree is implemented in an optimistic concurrency control BST. It shows the `tryLookup()` method described in Listing 7.1 with the additional code required to support CBTree marked by a □. The recursive search terminates when it cannot proceed (Lines 26-29) or when the key is found in the tree (Line 32). In all other cases the traversal needs to continue. If the traversal is in the process of skipping the current node (i.e., in Figure 4.1's terms, the current node is y while moving from z to x after finding that a rotation at z will not reduce the potential by ϵ) the `skip` parameter will be `TRUE` and `tryLookup` does not check if a rotation is required at the current node. Otherwise, before proceeding it calls `tryRotate` (Listing 7.3) which performs a rotation if one is required and returns a boolean indicating whether a rotation was done (Line 36).

When no rotation is needed, `tryRotate` returns `FALSE` and `tryLookup` proceeds as in Listing 7.1 by invoking a recursive call after validating the links to z and from z to y (Lines 42-44).

Alternatively, if `tryRotate` does perform a rotation then `tryLookup` returns `RETRY` to its caller (Line 38). Finally, `tryLookup` invokes itself recursively, toggling the `skip` parameter (Line 46). This ensures that after skipping a node, the next call will restructure the tree if possible.

Once the traversal terminates and the recursion unfolds, the appropriate child counters are incremented along the way if the key was found (Lines 48-53).

7.3 CBTree restructuring

Listing 7.3 details CBTree's restructuring process. The `tryRotate()` method computes the potential difference a rotation would cause and, if it is less than ϵ returns without performing a rotation (Lines 60-69. The potential difference is calculated using equations 4.1 and 4.2 from Section 4.2. (The actual implementation avoids the use of logarithms and compares the weight directly.)

If a rotation is required, locks are taken in parent-to-child order and the rotation is carried using Bronson et al.'s technique which perform the rotations in a way that allows concurrent traversals to safely proceed. Notice that when doing a single rotation, there is no need to lock y's right child r. If r is changed concurrently by another thread

T then r is the grandparent in T's rotation, for otherwise T must lock a node that is locked by the current rotation. Since the only field of r that the single rotation changes is the link to the parent, which T does not change, both rotations are safe and atomic.

Following the rotation, the node counters are adjusted to reflect the weight changes (Lines 78-79, 85-88).

```
23    Object tryLookup(key, z, dir, vers, skip) {
24      while (true) {
25        Node y = z.child(dir);
26        if (y == null) {
27          if (node.version != vers)
28            return RETRY;
29          return null; // not found
30        }
31        nextdir = compare(key, y.key)
32        if (nextdir == EQUAL) {
33 □        y.selfCnt++;
34          return y.value;
35        }
36 □      if (!skip and tryRotate(z, y, nextdir))
37        // validates links if it rotates
38 □        return RETRY;
39        long nextvers = y.version;
40        if (nextvers is changing)
41          waitUntilNotChanging(y);
42        else if (y == z.child(dir)
43                 and y not unlinked) {
44          if (z.version != vers)
45            return RETRY;
46          Object p=tryLookup(key,y,nextdir,nextvers,!skip);
47          if (p != RETRY) {
48 □          if (p != null) {  // found
49 □            if (nextdir == LEFT)
50 □              z.leftCnt++;
51 □            else
52 □              z.rightCnt++;
53 □          }
54            return p;
55          }
56    } } }
```

Listing 7.2: Main `lookup()` method

```
tryRotate(Node z, Node y, nextdir) {
  int doRot = NONE;
  if (z.left == y) {
    if (nextdir == LEFT) {
      if ($\Delta\Phi_1(z, y) < -\epsilon$)
        doRot = SINGLE;
    } else if (nextdir == RIGHT) {
      Node x = y.right;
      if (x != null and $\Delta\Phi_2(z, y, x) < -\epsilon$)
        doRot = DOUBLE;
    }
    if (doRot == NONE)
      return FALSE;
    Node grand = z.parent;
    synchronized (grand) { // locks grand
      if (grand.left == z or grand.right == z)
        synchronized (z) {
          if (z.left == y)
            synchronized (y) {
              if (doRot == SINGLE) {
                rotateRight(grand, z, y, y.right);
                z.leftCnt = y.rightCnt;
                y.rightCnt += z.selfCnt + z.rightCnt;
              } else {
                Node x = y.right;
                if (x != null)
                  synchronized (x) {
                    rotateRightOverLeft(grand, z, y, x);
                    z.leftCnt = x.rightCnt;
                    y.rightCnt = x.leftCnt;
                    x.rightCnt += z.selfCnt + z.rightCnt;
                    x.leftCnt += y.selfCnt + y.leftCnt;
    return TRUE; // signal rotation occurred
    } } } } }
  } else { // code for right child case omitted
    ...
  }
  // some validation failed; no rotation occurred
  return FALSE;
}
```

Listing 7.3: Self-adjusting decision. Code shown for when `y` is a left child; the right child case is symmetric. Once a decision to rotate is made, the nodes are rotated using an unmodified version of Bronson et al.'s rotation code (`rotateRightOverLeft()` and `rotateRight()`) which takes care of updating node version numbers so that tree traversals can safely execute concurrently to rotations.

7.4 CBTree insertion

Listing 7.4 shows the code of the `tryInsert()` method which determines whether to add a new node, update the value of an existing node, or continue navigating through the tree. We omit the details of the optimistic concurrency control, which are similar to those of the `lookup()` code. The `tryInsert` method first checks if the current key matches the desired key `k`. If so, it calls the `doInsert()` method to actually update the node (Lines 100-102). We do not show `doInsert`'s code as it is identical to the one in Bronson et al.'s implementation: it acquires a lock on the node and returns `RETRY` if the node has been concurrently unlinked. (This is the only relevant race, since no tree rotation can invalidate the fact that the correct node has been found.) If `doInsert` returns successfully the node's $selfCnt$ is updated (Line 104).

If the node's key does not match the desired key `k`, there are two cases. If the navigation cannot continue, a new node is added using the `addNode()` method which acquires lock on the node, performs the optimistic validation, and links a new node to the tree (Lines 109-111). (Again, we omit this standard code.). If the navigation does not terminate, `tryInsert` checks if a rotation is needed (Line 114), performs the hand-over-hand optimistic handoff and if successful invokes a recursive call (Lines 117-121). (As with a lookups, `tryInsert` returns `RETRY` after a rotation.) If the recursive call indicates the operation was successful, the appropriate child counter is incremented (Lines 122-125).

7.5 CBTree deletion

Our implementation of `delete` resembles Bronson et al.'s approach [8]. We `remove` an item v by first searching for it while restructuring the tree as in a `lookup`. If v is a leaf we unlink it from the tree. If v has a single child we remove v from the tree by linking v's parent to v's child. A `remove` of a node v with two children only marks v as `REMOVED`, turning it into a *routing node*. We adapt all operations to complete the semantics of `remove` as follows: (1) `lookup` fails if the desired item v is a routing node, (2) `insert` of an item found to be in a routing node unmarks the node, (3) any restructuring which changes a routing node to have fewer than two children unlinks it from the tree. With these changes, CBTree's space consumption thus remains linear in n, the number of non-routing nodes in the tree.

Upon unlinking v we *do not* decrease the weights along the path from v back to the root. In effect we remember v's history in case it is inserted back again. To do this we exploit the fact that a node's weights are maintained using three counters, $selfCnt$, $rightCnt$ and $leftCnt$, counting the total number of operations on v, items in v's right subtree and items in v's left subtree, respectively (Section 5.2). Thus a node's individual weight is always available, whereas its left and right counters may accumulate data about unlinked items.

Removals therefore do not decrease the tree's potential, so Theorem 2 holds for operation sequences containing `removes`.

We omit the code of the `remove()` and modifications supporting it in other operations since the restructuring `remove` performs during its search is identical to that of `insert` and `lookup`, while the handling of routing nodes is identical to that of Bronson et al.'s implementation [8].

7.6 Faster adaption to access pattern change

Here we discuss the extension that speeds up CBTree's response to a changing access pattern. We evaluate this extension in Section 8.6. To avoid having frequent nodes from the new distribution taking a lot of time until their counters are high enough to beat nodes that lost their popularity, we add an exponential decay function to the counters. The decay function is based on an external clock which ticks at a specified rate. The clock can be implemented in a number of ways: using a dedicated thread to periodically increment a global counter, relying on the operating system's time system call, or by reading from a hardware clock register. Accordingly, there are number of options for setting the clock rate: it can be a user-supplied parameter or a fixed value dictated by the OS or hardware.

We leave the optimal choice of clock implementation for future work. Here we describe our current implementation which uses a dedicated thread that increments a shared `globalClock` variable every *clockCycle* time interval, where *clockCycle* is set by the user (say to 10 milliseconds). All other threads only read `globalClock`, allowing it to be updated using a plain write.

Each counter value is divided by 2 every *clockCycle* time interval, as follows: Every node has an additional `timeStamp` field which contains the time when its counters have been updated last. When visiting a node and deciding whether to do a rotation, before comparing with any counter, if the `timeStamp` of the counter's node is smaller than the `globalClock` then each of the nodes' counters is divided by 2 for each *clockCycle* time that has elapsed since the `timeStamp`, and the node's `timeStamp` is then updated to the current time. Listing 7.5 shows the changes to `tryRotate()` to support this, and the `decayCounters()` method which performs the divisions.

We avoid concurrent decays of the same node by performing the counter divisions and the update of the node's `timeStamp` under the protection of the node's lock. To prevent these lock acquisitions from affecting the algorithm's scalability, we require the decay clock to advance at a rate considerably slower than the rate of operations, say on the order of milliseconds. (Clearly, a slow clock can be implemented from a fast clock.) Note that counter decays may still occur simultaneously to counter increments of other traversals, but our experiments show that in practice this isn't a problem.

```
  tryInsert(k, v, zParent, z, vers, skip)
  {
    int dir = compare(k, z.key)
    if (dir == EQUAL) {
      // doUpdate validates parent to node link if needed
      Object r = doUpdate(v, zParent, z);
      if (r != RETRY)
□       z.selfCnt++;
      return r;
    }
    while (true) {
      Node y = z.child(dir);
      if (y == null) { // new node
        y = addNode(z, vers, dir, k, v);   // validates parent to node link
        return null;
      } else {
        nextdir = compare(key, y.key)
□       if (!skip and tryRotate(z, y, nextdir))
        // validates links if it rotates
□         return RETRY;
        // omitted validation of parent to node link and
        // reading of node to child version into nextvers

        Object r = tryInsert(k, v, z, y, nextvers, !skip);
        if (r != RETRY) {
□         if (dir == LEFT)
□           z.leftCnt++;
□         else
□           z.rightCnt++;
          return r
        }
  } } }
```

Listing 7.4: Main `insert()` method

```
tryRotate(Node z, Node y, nextdir) {
  int now = globalClock;
  if (z.timeStamp < now)
    decayCounters(z, now);
  if (y.timeStamp < now)
    decayCounters(y, now);
  // the rest of tryRotate() remains
  // unchanged from Listing 7.3
}
decayCounters(Node node, int now) {
  synchronized (node) {
    // verify decaying still needed
    if (node.timeStamp < now) {
      int timeDiff = now − node.timeStamp;
      node.selfCnt = node.selfCnt >> timeDiff;
      node.rightCnt = node.rightCnt >> timeDiff;
      node.leftCnt = node.leftCnt >> timeDiff;
      node.timeStamp = now;
} } }
```

Listing 7.5: Counter decaying: we modify `tryRotate()` to decide if decaying is needed and call `decayCounters()` to decay them if so

7.7 LazyCBTree restructuring

Basically the implementation of LazyCBTree lookup, insert and delete function is similar to described above CBTree's implementation. The main differences are the restructuring of the tree which is done only once per operation and a simpler restructuring decision which doesn't require potential calculations.

```
Rebalance(Node parent, Node node) {
  if (parent.left == node) {
    nodePlusLeftCount = node.selfCnt + node.leftCnt;
    parentPlusRightCount = parent.selfCnt + parent.rightCnt;
    nodeRightCount = node.rightCnt;

    if (nodeRightCount >= parentPlusRightCount){
      Node grand = parent.parent;
      synchronized (grand) { // locks grand
        if (grand.left == parent) || (grand.right == parent)
          synchronized (parent) {
            if (parent.left == node)
              synchronized (node) {
                Node rightChild = node.right;
                if (rightChild != null)
                  synchronized (rightChild) {
                    rotateRightOverLeft(grand, parent, node, rightChild);
                    parent.leftCnt = rightChild.rightCnt;
                    node.rightCnt = rightChild.leftCnt;
                    rightChild.rightCnt += parentPlusRightCount;
                    rightChild.leftCnt += nodePlusLeftCount;
      }   }   } }
    } else if (nodePlusLeftCount > parentPlusRightCount) {
        Node grand = parent.parent;
        synchronized (grand) {
          if (grand.left == parent || grand.right == parent)
            synchronized (parent) {
              if (parent.left == node)
                synchronized (node) {
                  rotateRight(grand, parent, node, node.right);
                  parent.leftCnt = node.rightCnt;
                  node.rightCnt += parentPlusRightCount;
} } }   }   }
  // code for right child case omitted
}
```

Listing 7.6: Self-adjusting decision. Code shown for when `node` is a left child; the right child case is symmetric. Once a decision to rotate is made, the nodes are rotated using an unmodified version of Bronson et al.'s rotation code (`rotateRightOverLeft()` and `rotateRight()`) which takes care of updating node version numbers so that tree traversals can safely execute concurrently to rotations.

8. EXPERIMENTAL EVALUATION

In this section we compare the CBTree and LazyCBTree performance to the splay tree, treap [21] (a self-adjusting BST that we discussed in Section 3) and AVL algorithms. We base our implementations on Bronson et al.'s published source code.

8.1 Evaluation machines

Benchmarks are run on both a Sun UltraSPARC T2+ processor and on an Intel Xeon E7-4870 processor. The UltraSPARC T2+ (Niagara II) is a multithreading (CMT) processor, with 8 1.165 HZ in-order cores with 8 hardware strands per core, for a total of 64 hardware strands per chip. Each core has a private L1 write-through cache and the L2 cache is shared. The Intel Xeon E7-4870 (Westmere EX) processor has 10 2.40GHz cores, each multiplexing 2 hardware threads. Each core has private write-back L1 and L2 caches and a shared L3 cache.

On multicore architectures such as Intel's Xeon E7 a memory update occurs in a core's private cache, after the core acquires exclusive ownership of the cache line. Therefore, even relaxed counter maintenance can still degrade scalability. In CBTree and LazyCBTree implementations, when all cores frequently update the same counters (as happens at the top of the tree) each core invalidates a counter's cache line from its previous owner, who in turn had to take it from another core, and so on. On average, a core waits for all other cores to acquire the cache line before its write can complete. In contrast, on a multicore architecture like Sun's UltraSPARC T2 Plus all writes occur in a shared L2 cache, allowing the cores to proceed quickly: the L2 cache invalidates all L1 caches in parallel and completes the write.

8.2 Compared algorithms

Overall, we consider the following implementations: (1) **CB**, CBTree with decaying of node counters disabled, (2) **Splay**, Daniel Sleator's sequential top-down splay tree implementation [22] with a single lock to serialize all operations, (3) **Treap**, (4) **AVL**, Bronson et al.'s relaxed balance AVL tree [8] and (4) **LCB** LazyCBTree with decaying of node counters disabled. Because our dedicated adjuster technique applies to any self-adjusting BST, we include single adjuster versions of the splay tree and treap in our evaluation, which we refer to as **[Alg]OneAdjuster** for **Alg** $\in$ {Splay,Treap,CB,LCB}. In these implementations one dedicated thread alternates between doing lookups as in **Alg** for 1 millisecond and lookups without restructuring for t milliseconds ($t = 1$ on the UltraSPARC and $t = 10$ on the Intel; these values produced the best results overall). All other threads always run lookups without any restructuring. Insertions and removals are done as in **Alg** for all threads. Note that **SplayOneAdjuster** is implemented using Bronson et al.'s code to allow `lookups` to run concurrently with rotations performed by the adjuster.

8.3 Implementation details

To obtain the most meaningful results that minimize implementation artifact related differences, we base most implementations tested on Bronson et al.'s published source code. We make an exception for the splay tree, as we found that due to lock contention near the top of the tree, a coarse-grained locking implementation, with a one lock acquired for every operation, outperforms an implementation based on Bronson et al.'s code. We also tested semi-splaying, but we omit its results since we did not find a significant performance difference from splaying.

All algorithms are implemented in Java and are benchmarked using HotSpot Server JVM, build `1.7.0-ea-b137`. In all implementation a tree node contains a pointer to its item (a Java object); the data is not stored in the node. Item comparisons are done using the Java `Comparable` interface, which item objects must support. Our tests use `Integer` and `String` objects, where comparisons are done in lexicographic order. We avoid having garbage collection trigger during the measurements by setting the Java heap size to 2 GB, which exceeds the benchmarks' maximum memory consumption. (We verified that indeed no garbage collection occurred during the measurements.) Unless stated otherwise, reported results are averages of three 10-second runs with little variance, run on an otherwise idle machine.

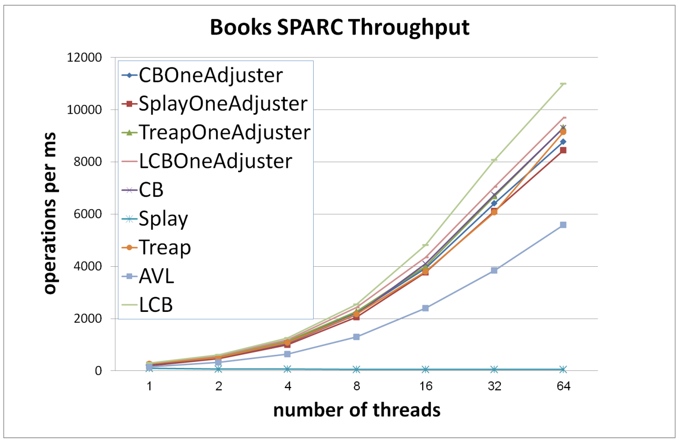
8.4 Realistic workloads

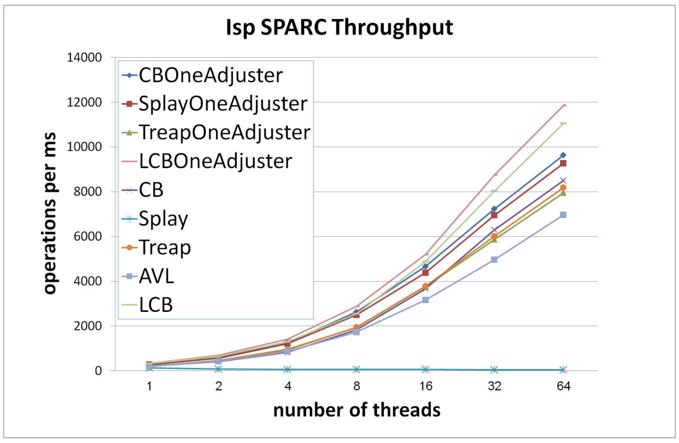
Here the algorithms are tested on access patterns derived from real workloads: (1) **books**, a sequence of 1,800,825 words (with 31,779 unique words) generated by concatenating ten books from Project Gutenberg [1], (2) **isp**, a sequence of 27,318,568 IP addresses (449,707 unique) from packets captured on a 10 gigabit/second backbone link of a Tier1 ISP between Chicago, IL and Seattle, WA in March, 2011 [17], and (3) **youtube**, a sequence of 1,467,700 IP addresses (39,852 unique) from YouTube user request data collected in a campus network measurement [27]. As the traces we obtained are of item sequences without accompanying operations, in this test we use only `lookup` operations on the items in the trace, with no `inserts` or `removes`. To avoid initialization effects each algorithm starts with a maximum balanced tree over the domain, i.e., where the median item is the root, the first quartile is the root's left child, and so on. Each thread then repeatedly acquires a 1000-operation chunk of the sequence and invokes the operations in that subsequence in order, wrapping back to the beginning of the sequence after the entire sequence has been performed.

Table 8.1 shows the average number of nodes traversed and rotations done by each operation on the Sun UltraSPARC machine. As these are algorithmic rather than implementation metrics, results on the Intel are similar and thus omitted. Figure 8.1 shows the number of operations completed by all the threads during the duration of the test (throughput).

	AVL	Splay	Treap	CBTree	Splay	Treap	CBTree
						Single adjuster	
Average path length							
books	17.64	10.63, 12.06	11.19	**9.54, 8.64**	11.71	11.43	**10.13, 11.06**
isp	17.86	9.09, 12.89	14.64	**11.13**	13.39	14.46, 15.1	**12.33, 13.25**
youtube	14.35	8.52, 13.31	15.75	**11.81**	13.47	15.84	**12.27**
Rotations per operation							
books	0	9.16	< 0.01	< **0.01**	2.95	0.09	**0.02**
isp	0	8.09, 10.45	0.03	**0.01**	2.65, 3.10	0.25	**0.01**
youtube	0	7.52, 10.73	< 0.01	< **0.01**	3.17	0.11	**0.03**

Tab. 8.1: Average path length and number of rotations for a single thread and 64 threads. When the single thread result, r_1, significantly differs from the 64 threads result, r_{64}, we report both as r_1, r_{64}. To reduce overhead, data is collected by one representative thread, who is the adjuster in the single adjuster variants.





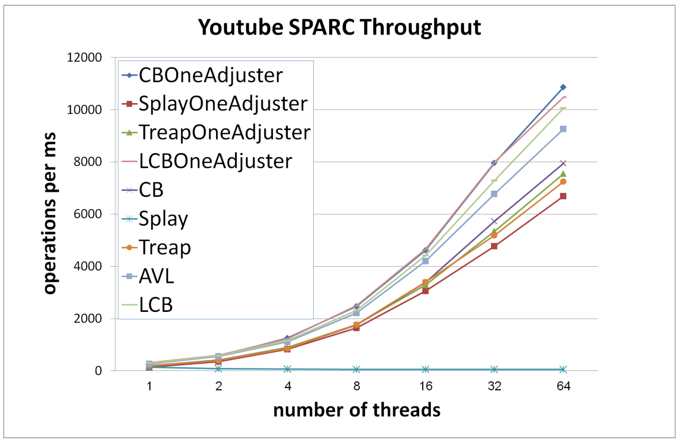
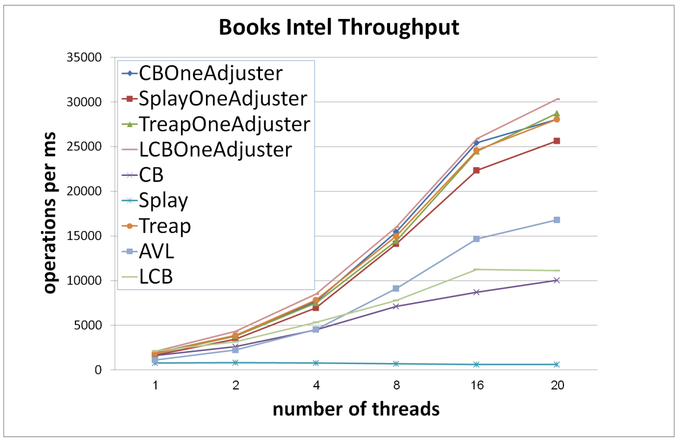
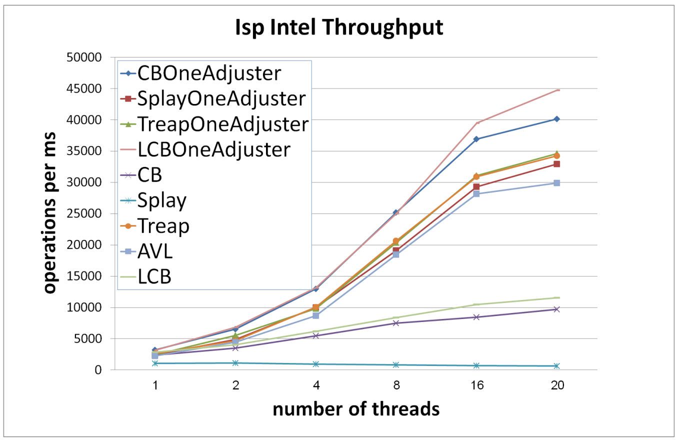


Fig. 8.1: Test sequence results on Sun UltraSPARC T2+ (up to 64 hardware threads).

CBTree obtains the best path length, however this does not translate to the best overall performance due to the overhead of computing potential differences. On the UltraSPARC machine CBTree scales well because counter updates do not cause serializing behavior on this architecture, as all writes are performed in the shared L2 cache. Counter updates do add overhead and therefore, despite scaling well, CBTree on the UltraSPARC obtains lower throughput than some of the other algorithms. On the Intel Xeon E7 concurrent updates of the shared counters serialize the threads due to cache coherency (see Section 5.3), and thus CBTree scales poorly. In fact, on the Intel





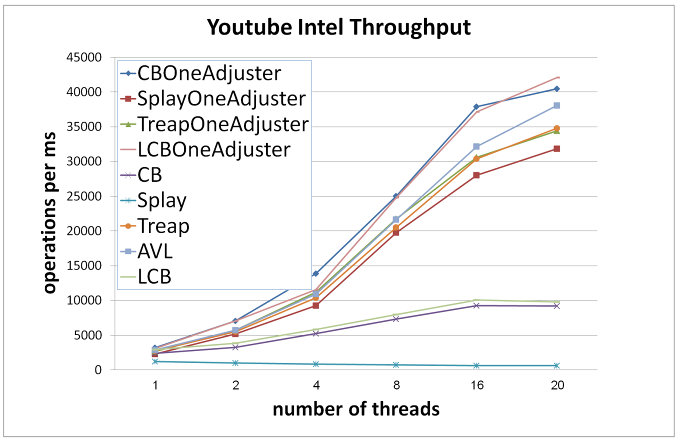


Fig. 8.2: Test sequence results on Intel Westmere EX (up to 20 hardware threads).

even the AVL tree, which employs read-only `lookups`, outperforms CBTree despite the AVL tree's poor path length.

CBOneAdjuster addresses both of CBTree's overhead and cache coherency serialization problems by removing the shared counter updates, making **CBOneAdjuster** the best overall performer on both architectures. For example, running the **books** sequence on the UltraSPARC machine, **CBOneAdjuster** outperforms Treap, the next best self-adjusting algorithm, by about 1.44× at 64 threads. The treap obtains its performance because with high probability, an operation only updates its target node.

However, this results in suboptimal path lengths, and also prevents the treap from seeing much benefit due to the single adjuster technique.

While the AVL tree scales well, its lack of self-adjusting leads to suboptimal path lengths and performance. On **books**, for example, **CBOneAdjuster** outperforms AVL by 1.75× at 64 threads on the UltraSPARC machine.

The splay tree's path degrades as concurrency increases because interleaving of thread operations changes the locality property of the access pattern. Recall that each thread works on a 1000-operation chunk of the trace; thus two operations in different chunks (i.e., far apart in the original trace) may arrive close together in a concurrent setting. Indeed, we obtain a similar path length to the 64 thread execution in a sequential execution of the splay tree by reordering the sequence as follows: $1, 1000, 2000, \ldots, 64000, 2, 1001, \ldots, 64001, \ldots$. The other self-adjusting BSTs are more robust to this problem as they focus on the overall frequency of item accesses.

In addition, the splay's tree coarse lock prevents it from translating its short path length into actual performance. Applying our single adjuster optimization which allows readers to run concurrently while benefiting from the adjuster's splaying, resolves this problem and yields a scalable algorithm with a significant higher throughput. Notice that despite obtaining comparable path lengths to **CBOneAdjuster**, the **SplayOneAdjuster** does $> 100\times$ more rotations than **CBOneAdjuster**, which force the concurrent traversals to retry. Moreover, the locality property of the splay tree enables the adjuster thread to modify the tree to best fit its own input. However, the inputs of the other threads don't come into consideration, thus causing these threads to travel longer paths. For example on **isp** benchmark with one Splay thread doing adjusting all the time and 63 other threads do read-only finds, the adjuster reports average path length of 9.5, while non adjusters report average path length of 15.7! As a result, **CBOneAdjuster** outperforms **SplayOneAdjuster**.

Since LazyCBTree checks its rotations condition only once per operation it always outperforms CBTree. However on Intel machine, LazyCBTree has relatively small advantage, since updating of the counters is their common Achilles heel. In OneAdjuster the advantage of LazyCBTree over CBTree almost disappears, since the performance differences of the adjusters themselves are negligible.

8.5 Zipf skewed usage pattern

In this section we evaluate the performance of the algorithms on synthetic skewed workloads following a Zipf distribution[1] [28]. We measure throughput and path length of the algorithms for different Zipf skew values over a domain of 130K items. Figure 8.3 shows the results from the SPARC and Intel machines at maximum concurrency level. Here we report results of **CBAdapt**, which is CBTree where we decay node counters every 1 second. As before the test starts with a balanced tree over the domain. Each thread performs a combination of 9% `insert()`, 90% /`lookup()` and 1% /`remove()` operations. The `insert` and `lookup` operations access items following the Zipf distribution; `removed` items are selected uniformly. This results in faster return of frequent nodes after they have been removed.

While the splay tree's path length becomes shorter as the skew increases, its throughput remains constant since it is limited by the coarse-grained lock. In contrast, the AVL tree has no sensitivity to the skew as it is not self-adjusting; its path length and throughput remain constant. On the other hand, the treap's and CBTree's path length decreases with the skew. Beyond skew level 0.6 (80% accesses to 55% of the items) **CBOneAdjuster** outperforms the AVL tree.

[1] In an N-item Zipf distribution with skew γ, the k-th most frequent item appears with frequency $\frac{1}{k^\gamma} / \sum_{n=1}^{N} \frac{1}{n^\gamma}$.

LazyCBTree again outperforms CBTree. Though LazyCBTree creates trees with only slightly better paths lengths, avoidance of multiple checks enables it much better performance on the Sun UltraSPARC. In OneAdjuster the LazyCBTree and CBTree have similar performance like in 8.4.

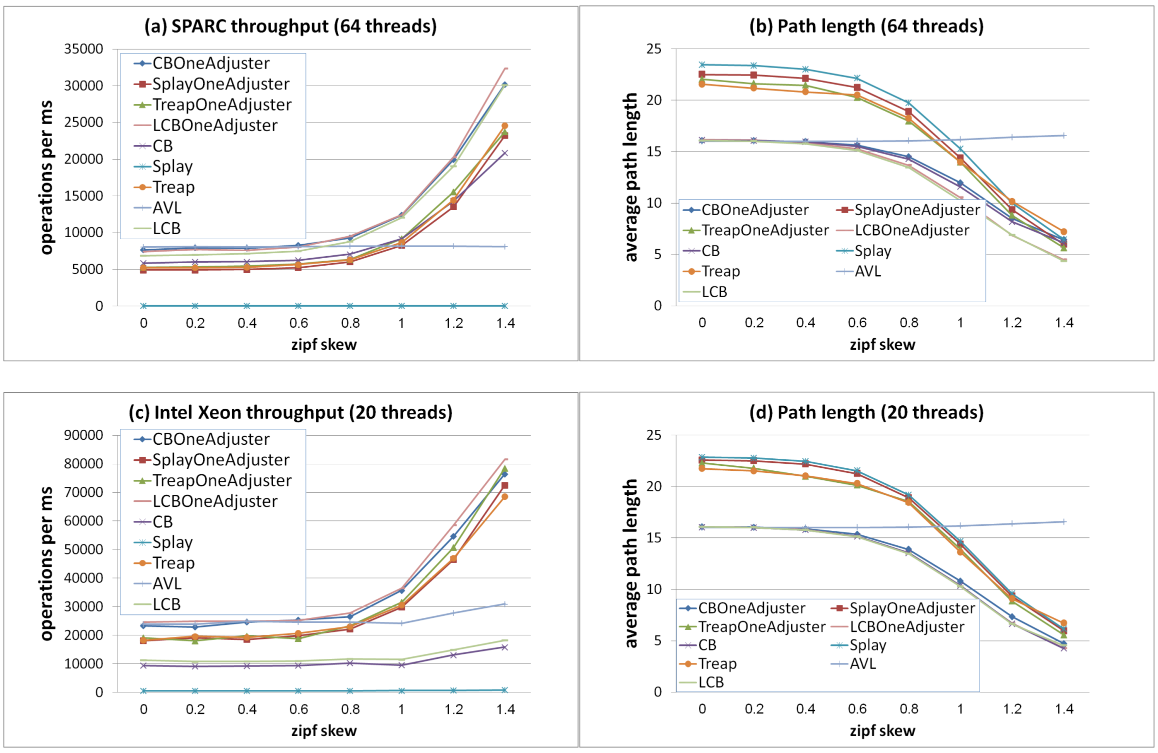


Fig. 8.3: Performance at maximum concurrency as function of Zipf skew. **Left:** SPARC (64 threads). **Right:** Intel Westmere EX (20 threads).

8.6 *Adapting to changing usage pattern*

Here we examine how the algorithms adjust to changes in the usage pattern. We start with an empty tree on the Sun UltraSPARC and let 32 threads run for 15 seconds while performing only `insert()` operations on items sampled from a Zipf distribution with skew of 0.8 (80% accesses to 37% of the items). Then, in the last 30 seconds we switch to a new sampling of the Zipf distribution with the same skew. Recall that once the tree fills with all the items in the key range, `insert()` operations behave like a `lookup()`.

Figures 8.4a-8.4b compare the results of the implementations as well as **CBAdapt**, which is CBTree where we decay node counters every 1 second. In the first 15 seconds CBTree and **CBAdapt** obtain roughly the same throughput, both outperforming all other algorithms by at least 10%. After the distribution switch **CBAdapt** is the quickest to recover, remaining the best performer. The treap and CBTree both adapt slowly to access pattern change. The AVL tree is not self-adjusting and thus once the distribution changes it is stuck with a tree that is not suitable for the new distribution and its performance deteriorates.

In Figures 8.4c-8.4d we evaluate the counter decay mechanism. Here **CBAdapt**(t) stands for CBTree where counters are decayed every t milliseconds. As counters are decayed more aggressively the CBTree's path length adapts to the distribution switch more quickly (Figure 8.4d), with **CBAdapt**(100), the most aggressive variant, obtaining the highest throughput. On the other hand, frequent counter decays add extra overhead and decrease the algorithm's overall performance. This is evident from the fact that in the first 15 seconds, all variants have the same average path length, and yet those that aggressively decay their counters have lower overall performance.

8.7 Performance under different operation mixes

In this section we evaluate the scalability of the implementations under different operation mixes. For the self-adjusting algorithms, we test both with and without the single-adjuster optimization and report the best performing variant. Figure 8.5 shows throughput results on the UltraSPARC T2+ machine for various mixes. Recall that this processor supports 64 hardware threads, and so the 128 threads results show the performance effect of OS context switches that are caused when more threads than available hardware threads are running. Even though counter updates are non-atomic and may be disrupted by context-switches, in practice context switches have the same effect on CBTree as on the other algorithms.

With the single-adjuster optimization, all threads but the adjuster have read-only `lookups`, but their insertions and removals follow the original algorithm. Consequentially, the single adjuster variants' performance depends on the ratio of `lookups`. For **SplayOneAdjuster**, even 10% update operation cause several contention at the tree top and prevent scaling. In contrast, **CBOneAdjuster** achieves the best throughput when there are at least 70% `lookups`. However, as the ratio of `lookups` decreases, **CBOneAdjuster**'s margin from next best implementation, the AVL tree, decreases

Figure 8.6 shows throughput results on the Intel Xeon machine for various mixes. Here our single-adjuster optimization is not enough. Since continuous update of counters by various threads is the bottle-neck on this architecture, even small percentage of inserts reduces the performance. Though OneAdjuster versions outperform AVL on 100% lookups test by far, their advantage when 1% of inserts performed is very small. In scenario with 90% lookups AVL already outperforms the other algorithms and the gap grows further on test with 70% lookups. Bottom line our algorithm does't work well on Intel Xeon like architecture with operations mix.

vma benchmark represents process virtual memory maintenance operations recorded during the execution of the `dedup` program from the PARSEC benchmark suite [6]. **vma** sequence length is 1,017,394 and it consist of 7674 unique items. Unlike other realistic workloads in 8.4, **vma** also performs insertions and deletion. As stated above and seen in 8.7, single-adjuster versions on Intel Xeon don't provide good performance on this benchmark.

8.8 Sensitivity to lost counter updates

The purpose of the test here is to investigate the robustness of the algorithm in the face of erroneous changes to the counters due to their unprotected updates. In this test, every thread every *errorFrequency* operations injects an error into one of the node's counters. $leftCnt$ and $rightCnt$ are disrupted three times more than $selfCnt$ since they are incremented more by the algorithm. The injected errors reduced the counter values by $errorValue$, which was taken either -2000 or -50000. Figure 8.8 depicts the resulting throughput, which shows that with $errorValue < 2000$ there is almost no effect.

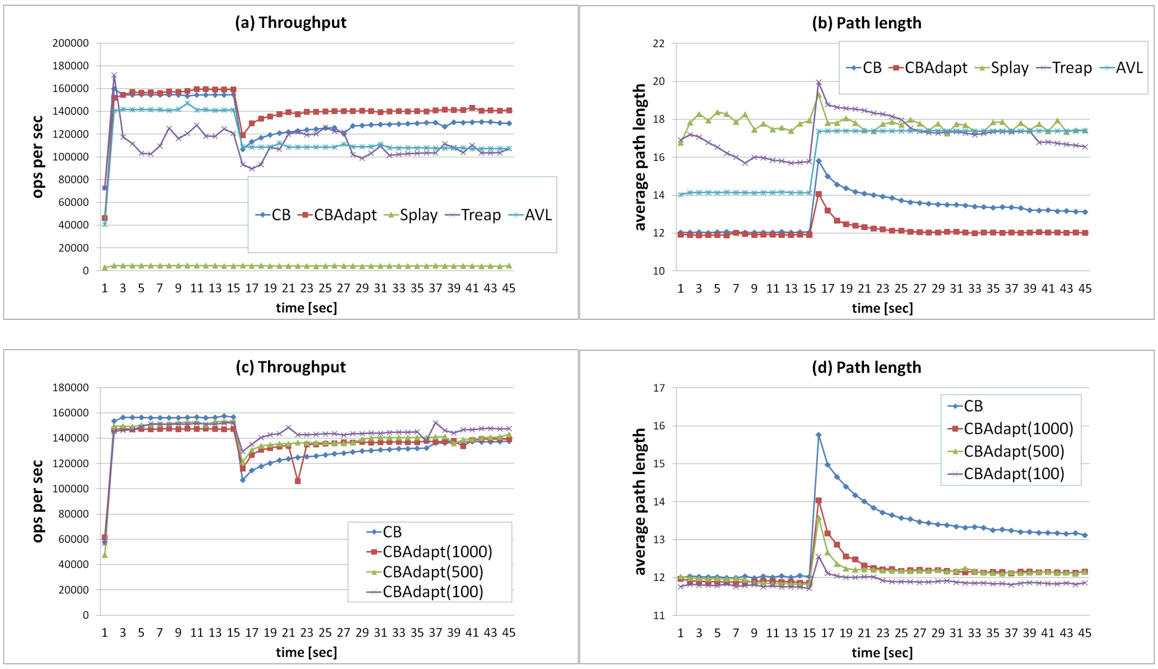
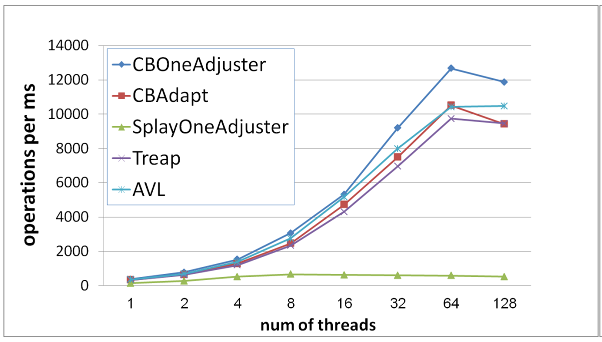
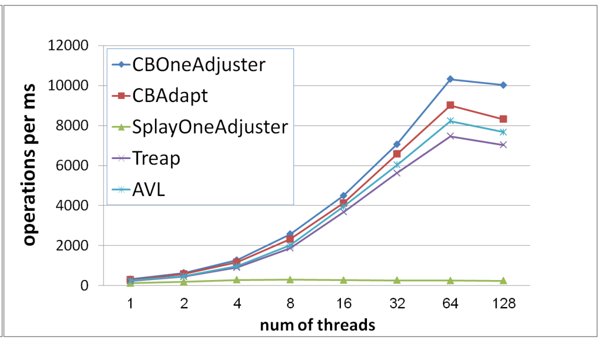


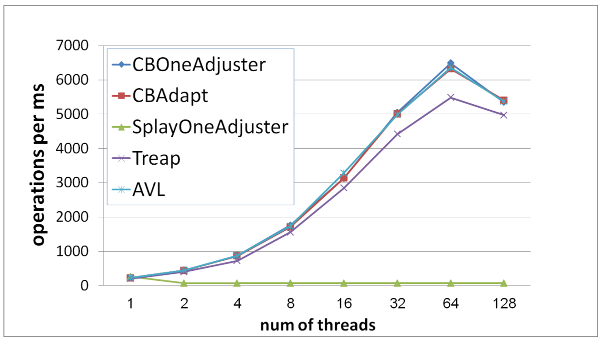
Fig. 8.4: Adapting to a varying the usage pattern (32 threads execution on the Sun Ultra-SPARC). For the first 15 seconds requests follow a Zipf distribution with skew 0.8. Afterwards a different distribution with the same skew is used. **Top:** comparing all algorithms. **Bottom:** Impact of CBTree counter decay aggressiveness.



(a) 9% `put()`, 1% `remove()`, 90% `get()`

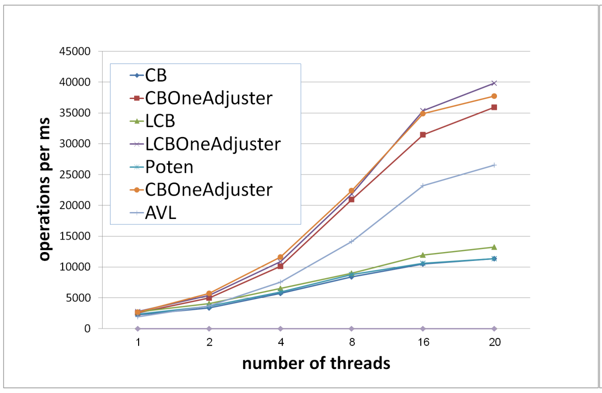


(b) 20% `put()`, 10% `remove()`, 70% `get()`

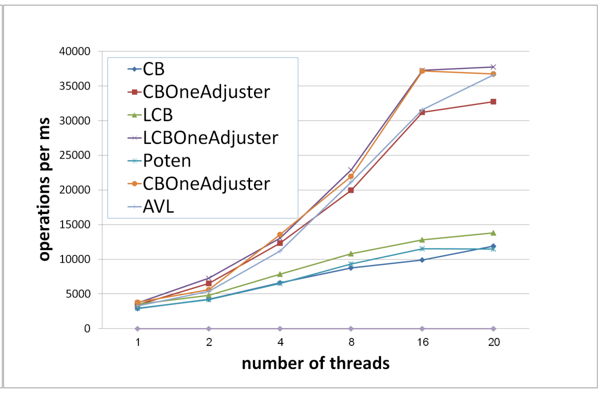


(c) 50% `put()`, 50% `remove()`, 0% `get()`

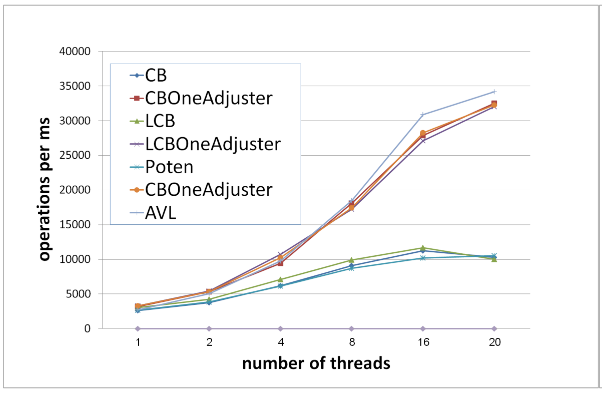
Fig. 8.5: Scalability test on SPARC: Performance on a 130K nodes tree with different operation mix as a function of the number of threads (processor has 64 hardware threads so at 128 software threads there is context switching). Zipf skew is 0.94 (80% accesses to 20% of the items). `remove()` operations items are uniformly distributed.



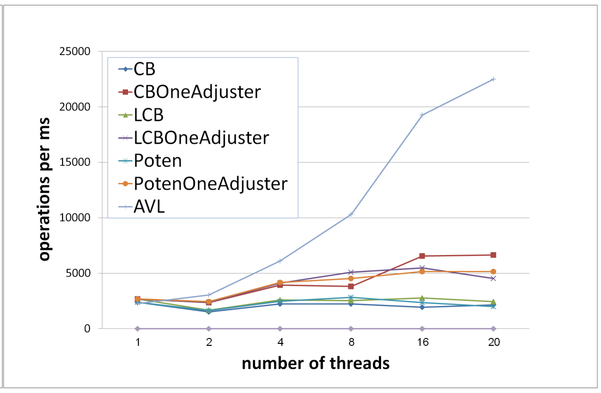
(a) 0% `put()`, 0% `remove()`, 100% `get()`



(b) 1% `put()`, 0% `remove()`, 99% `get()`



(c) 9% `put()`, 1% `remove()`, 90% `get()`



(d) 20% `put()`, 10% `remove()`, 70% `get()`

Fig. 8.6: Scalability test on Intel: Performance on a 130K nodes tree with different operation mix as a function of the number of threads. Zipf skew is 0.94 (80% accesses to 20% of the items). `remove()` operations items are uniformly distributed.

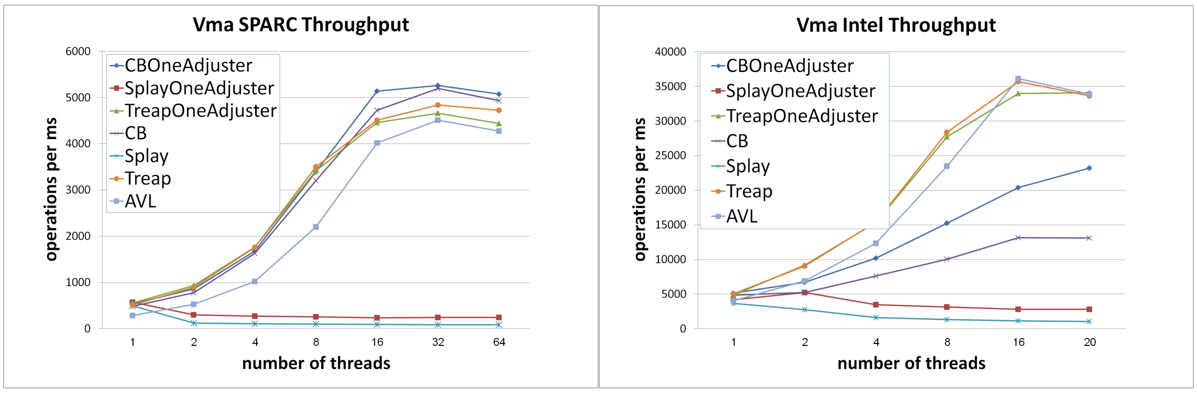


Fig. 8.7: vma test sequence results on Sun UltraSPARC T2+ (up to 64 hardware threads) and on Intel Westmere EX (up to 20 hardware threads)

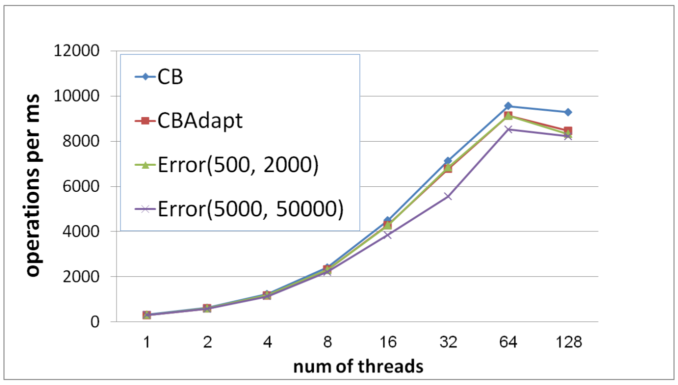


Fig. 8.8: Error injection test. This test simulates the situation in which a scheduler makes context-switches during the increment of the counter. This may result in lost counter updates. The performance on a tree with 130K nodes and operation mix of 9% `insert()`, 1% `remove()`, 90% `get()`. Zipf skew is 0.94. In Errors(500, 2000) and Errors(5000, 50000) once every 500 and 5000 operations respectively some counter value is decreased by 2000 and 50000 respectively. Even such a drastic out of date counter reduction has only small effect on performance.

9. DIRECTIONS FOR FUTURE WORK

CBTree and LazyCBTree are still have some drawbacks that might be interesting to solve. As we have showed CBTree has provable ideal asymptotical time bound, however it looses to LazyCBTree in the performance. It would be interesting to find an algorithm which will combine CBTree's ideal asymptotical time bound with LazyCBTree's performance.

Both CBTree and LazyCBTree slowly adjust the tree due to counters updates and they indeed prioritize frequently used item which are later found more quickly. However, CBTree and LazyCBTree are lack of locality and recency considerations as splay tree has. It would be interesting to provide an algorithm which will have good behavior on frequency based inputs like zipf, and will make almost no rotations. However, on the locality based input as in the Youtube benchmark, where items are frequent during some small time interval, the algorithm will quickly move them towards the top of the tree as splay tree does.

As we found the algorithm does't work well on Intel Xeon like architecture with operations mix. It would be most interesting to develop an algorithm that will overcome this problem.

BIBLIOGRAPHY

[1] Project Gutenberg. `http://www.gutenberg.org/`.

[2] M.F. Arlitt and C.L. Williamson. Internet web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, Oct 1997.

[3] Jean-Loup Baer. Weight-balanced trees. In *American Federation of Information Processing Societies: 1975 National Computer Conference*, AFIPS '75, pages 467–472, New York, NY, USA, 1975. ACM.

[4] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. In *Readings in database systems*, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[5] Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased 2-3 trees. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science*, FOCS '80, pages 248–254, Washington, DC, USA, 1980. IEEE Computer Society.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[7] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM '99, pages 126–134, March 1999.

[8] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, New York, NY, USA, 2010. ACM.

[9] L. Cherkasova and Minaxi Gupta. Analysis of enterprise media server workloads: access patterns, locality, content evolution, and rates of change. *IEEE/ACM Transactions on Networking*, 12(5):781–794, Oct 2004.

[10] Maureen Chesire, Alec Wolman, Geoffrey M. Voelker, and Henry M. Levy. Measurement and analysis of a streaming-media workload. In *Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, USITS'01, Berkeley, CA, USA, 2001. USENIX Association.

[11] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 161–170, New York, NY, USA, 2012. ACM.

[12] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[13] J. Feigenbaum and R. E. Tarjan. Two new kinds of biased search trees. *Bell System Technical Joural*, 62:3139–3158, December 1983.

[14] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. YouTube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 15–28, New York, NY, USA, 2007. ACM.

[15] S. Hanke, Th. Ottmann, and E. Soisalon-soininen. Relaxed balanced red-black trees. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, pages 193–204. Springer Verlag, 1997.

[16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12:463–492, July 1990.

[17] kc claffy, Dan Andersen, and Paul Hick. The caida anonymized 2011 internet traces. `http://www.caida.org/data/passive/passive_2011_dataset.xml`.

[18] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[19] A. Mahanti, C. Williamson, and D. Eager. Traffic analysis of a web proxy caching hierarchy. *IEEE Network*, 14(3):16–23, May/Jun 2000.

[20] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33:668–676, June 1990.

[21] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996. 10.1007/s004539900061.

[22] Daniel Dominic Sleator. Splay tree implementation. `http://www.link.cs.cmu.edu/splay`.

[23] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, July 1985.

[24] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet <easurement*, IMC '04, pages 41–54, New York, NY, USA, 2004. ACM.

[25] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[26] Hongliang Yu, Dongdong Zheng, Ben Y. Zhao, and Weimin Zheng. Understanding user behavior in large-scale video-on-demand systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '06, pages 333–344, New York, NY, USA, 2006. ACM.

[27] Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. Watch global, cache local: YouTube network traffic at a campus network - measurements and implications. *Proceeding of the 15th SPIE/ACM Multimedia Computing and Networking Conference*, 6818:28, 2008.

[28] George Kingsley Zipf. *Human Behavior and Principle of Least Effort: An Introduction to Human Ecology*. Addison Wesley, 1949.