CBTree: A Practical Concurrent Self-adjusting Search Tree

Yehuda Afek[1], Haim Kaplan[1], Boris Korenfeld[1],
Adam Morrison[1], and Robert E. Tarjan[2]

[1] Blavatnik School of Computer Science, Tel Aviv University
[2] Princeton University and HP Labs

Abstract. We present the CBTree, a new *counting-based* self-adjusting binary search tree that, like *splay trees*, moves more frequently accessed nodes closer to the root. After m operations on n items, c of which access some item v, an operation on v traverses a path of length $\mathcal{O}(\log \frac{m}{c})$ 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, the CBTree performs rotations infrequently (an amortized subconstant $o(1)$ per operation if $m \gg n$), mostly at the bottom of the tree. As a result, the CBTree scales with the amount of concurrency. We adapt the 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.

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 [1,2,3,4]. Therefore, in this paper we develop a concurrent data structure that completes operations on popular items faster than on ones accessed infrequently, 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.

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., [5,6]) 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 do not provide optimal performance for skewed and changing usage patterns.

Unlike these BSTs, 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* [7], which moves each accessed node to the root using a sequence of rotations called *splaying*. The amortized access time of a splay

M.K. Aguilera (Ed.): DISC 2012, LNCS 7611, pp. 1–15, 2012.

tree for an item v which is accessed $c(v)$ times in a sequence of m operations is $\mathcal{O}(\log \frac{m}{c(v)})$, asymptotically matching the information-theoretic optimum [8].

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. We discuss the limitations of some other sequential self-adjusting BSTs in Sect. 2. 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 this paper, we present a *counting-based tree*, CBTree for short, a self-adjusting BST that scales with the amount of concurrency, and has performance guarantees similar to the splay tree. The CBTree maintains a *weight* for each subtree S, equal to the total number of accesses to items in S. The CBTree operations use rotations in a way similar to splay trees, but rather than performing them at each node along the access path, decisions of where to rotate are based on the weights. The 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, c of which access item v, an operation on v takes time $\mathcal{O}(1 + \log \frac{m}{c})$.

The CBTree's crucial performance property is that it performs only a *sub-constant* $o(1)$ amortized number of rotations per operation, so most CBTree operations perform few if any rotations. This allows the CBTree to scale with the amount of concurrency 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.

The CBTree replaces most of the rotations splaying does with counter updates, which are local and do not change the structure of the tree. To translate the 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 the 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 counters 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 the CBTree's counters over time so that it responds faster to a change in the access pattern.

One can implement the CBTree easily on top of any concurrent code for doing rotations atomically, because the CBTree restructures itself by rotations as does any other BST. Our implementation uses Bronson et al.'s optimistic BST concurrency control technique [5]. We compare our CBTree implementation with other sequential and concurrent BSTs using real workloads, and show that the CBTree provides short access paths and excellent throughput.

2 Related Work

Treaps. A treap [9] is a BST satisfying the *heap property*: each node v also has a priority which is maximal in its subtree. An access to node v generates a random number r which replaces v's priority if it is greater than it, after which v is rotated up the treap until the heap property is restored. Treaps provide *probabilistic* bounds similar to those of the CBTree: node v is at expected depth $\mathcal{O}(\log \frac{m}{c(v)})$ and accessing it incurs an expected $\mathcal{O}(1)$ rotations [9]. Since the treap's rotations are driven by local decisions it is suitable for a concurrent implementation. However, we find that in practice nodes' depths in the treap vary from the expected bound. Consequentially, CBTrees (and splay trees) have better path length than treaps (Sect. 5).

Binary Search Trees of Bounded Balance. Nievergelt and Reingold [10] described how to keep a search tree balanced by doing rotations based on subtree sizes. Their goal was to maintain $O(\log n)$ height; they did not consider making more popular items faster to access.

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 discuss these variants extensively [11]. 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 [11,12], for example, every operation is implemented using global tree splits and joins, and items are only stored in the leaves. The CBTree's rotations are the same as those in Baer's weight-balanced trees [13], but the CBTree uses different rules to decide when to apply rotations. Our analysis of CBTrees 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. [5,6]. Ellen et al. proposed a nonblocking concurrent BST [14]. Their tree is not balanced, and their focus was obtaining a lock-free BST. Crain et al.'s recent transaction-friendly BST [15] is a balanced tree in which a dedicated thread continuously scans the tree looking for balancing rule violations

that it then corrects. Unlike the CBTree's adjuster, this thread does not perform other BST operations. Ordered sets and maps can also be implemented using skip lists [16], which are also not self-adjusting.

3 The CBTree and Its Analysis

3.1 Splaying Analysis Background

The CBTree's design draws from the analysis of semi-splaying, a variant of splaying [7]. 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 examines the next 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 Fig. 1. Following the rotation(s) it continues from the node which replaces the current node's grandparent in the tree (in Fig. 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.

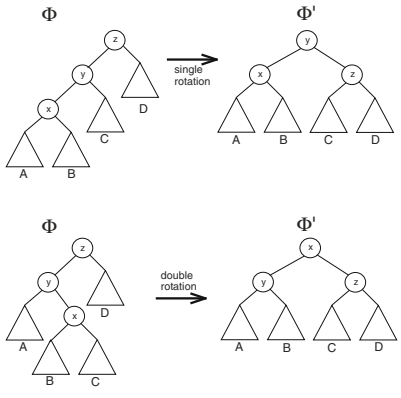

Fig. 1. Semi-splaying restructuring: current node is x, and the next two nodes on the path to the root are y and z. The case when y is a right child is symmetric.

To analyze the amortized performance of splaying and semi-splaying Sleator and Tarjan use the potential method [17]. The potential of a splay tree is defined based on 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 [7]:

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 Fig. 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.

[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 analysis uses this lemma to show that the amortized time of semi-splaying node v in a tree rooted at *root* is $\mathcal{O}(r(root) - r(v)) = \mathcal{O}(\lg(W(root)/W(v)) + 1)$. 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 $c(v)$ be the number of operations on v in the sequence. Using Lemma 1 with this weight assignment Sleator and Tarjan proved the following [7]:

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*

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

where $c(v)$ is the number of times v is accessed in the sequence.

Hereafter we say that $\mathcal{O}(\lg(m/c(v)) + 1)$ is the *ideal access time* of v.

3.2 The 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 performed on v until now, 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 on 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 Fig. 1. If the child of y along the path is a right child we may perform a double rotation as in the bottom part of Fig. 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 a positive constant $\epsilon \in (0, 2)$, 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.

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]

Delete: We `delete` 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. If v has two children we only mark it as deleted. We adapt all operations so that any restructuring which makes a deleted node a leaf or a single child unlinks it from the tree. With these changes, CBTree's space consumption remains linear in n, the number of non-deleted nodes in the tree. If we are willing to count failed `lookup`s as accesses, we can update the counters top-down instead of bottom-up.

Computing Potential Differences: To decide whether to perform a rotation, an operation needs to compute $\Delta\Phi$, the potential difference resulting from the 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 Fig. 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

$$\Delta\Phi = \lg(C(z) + W(y.right) + W(z.right)) - \lg W(y). \tag{1}$$

For a double rotation, an analogous derivation yields

$$\begin{aligned}\Delta\Phi = &\lg\left(C(z) + W(x.right) + W(z.right)\right) + \\ &\lg\left(C(y) + W(y.left) + W(x.left)\right) - \\ &\lg W(y) - \lg W(x).\end{aligned} \tag{2}$$

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

Note that our implementation works with the weights directly by computing $2^{\Delta\Phi}$ using logarithmic identities and comparing it to $2^{-\epsilon}$.

[3] A failed `insert()` can change auxiliary information associated with v.

An alternative rule for deciding when to do a rotation is to rotate when $W(z)/W(x) < \alpha$, where α is a constant less than two. This rule is simpler to apply, simpler to analyze, and more intuitive than rotating when the potential drops by at least ϵ, but we have not yet had time to try it in experiments.

3.3 Analysis

In this section we consider only *successful* lookups, that is, lookups of items that are in the tree. For simplicity, we do not consider deleting an item v and then later inserting it again, although the results can be extended by considering such an item to be a new item. We prove that Theorem 1 holds for the CBTree.

Theorem 2. *Consider a sequence of m operations on n distinct items, $v_1, \ldots, v_n$, starting with an initially empty CBTree. Then the total time it takes to perform the sequence is*

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

where $c(v)$ is the number of times v is accessed in the sequence.

An 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 first prove that an operation on node v, applied to a CBTree with weights $W(u)$ for each node u, traverses $\mathcal{O}(\lg(W/C(v))$ edges in between rotations, where $W = W(root)$ is the weight of the entire tree and $C(v)$ is the individual weight of v at the time the operation is performed (Lemma 2). From this we obtain that the time spent traversing edges between rotations is $\mathcal{O}(c(v) + c(v) \lg \frac{m}{c(v)})$ (Lemma 3). 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 $\mathcal{O}(n + n \ln \frac{m}{n}) = \mathcal{O}(m)$.

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 $\mathcal{O}(\lg(W/C(v)) + 1)$, 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 at the end of the operation (i.e., just before v is unlinked if the operation is a `delete`) consists of d pairs of edges (z, y) and (y, x) that the operation skipped between rotations, and possibly a single final edge if the operation could look ahead only one node 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

$$2(r(z) - r(x)) \geq 2 + \Delta\Phi > 2 - \epsilon. \tag{3}$$

Define $\delta = 1 - \epsilon/2$. Then Equation (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

$$d < \frac{r(root) - r(v)}{\delta} = \mathcal{O}\left(\lg \frac{W(root)}{W(v)}\right) = \mathcal{O}\left(\lg \frac{W}{C(v)}\right).$$

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

We now show that Lemma 2's bound matches that of the splay tree.

Lemma 3. *The total time spent traversing edges between rotations in all the operations that access v is $\mathcal{O}(c(v) + c(v) \lg \frac{m}{c(v)})$.*

Proof. The time spent traversing edges between rotations is 1 plus the number of edges traversed. Because the CBTree's weight is at most m throughout the sequence, and v's individual weight after the k-th time it is accessed is k, Lemma 2 implies that the time spent traversing edges between rotations over all the operations accessing v is $c(v) + \mathcal{O}\left(\sum_{j=1}^{c(v)} \lg \frac{m}{j}\right)$. By considering separately the cost of the final half of the operations, the quarter before it, and so on, we bound this as follows:

$$\begin{aligned} & c(v) + \mathcal{O}\left(\frac{c(v)}{2} \lg \frac{m}{c(v)/2} + \frac{c(v)}{4} \lg \frac{m}{c(v)/4} + \dots\right) \\ =& c(v) + \mathcal{O}\left(c(v) \lg \frac{m}{c(v)} + c(v)\left(\frac{\lg 2}{2} + \frac{\lg 4}{4} + \dots\right)\right), \end{aligned}$$

which is $\mathcal{O}\left(c(v) + c(v) \lg \frac{m}{c(v)}\right)$ because $\sum_{k=1}^{\infty} \frac{k}{2^k} = 2$. □

The following lemma, proved in the extended version of the paper [18], bounds the number of rotations.

Lemma 4. *In a sequence of m operations starting with an empty CBTree, we perform $O(n + n \ln \frac{m}{n}) = \mathcal{O}(m)$ rotations, where n is the number of insertions.*

4 The Concurrent CBTree

We demonstrate the CBTree using Bronson et al.'s optimistic BST concurrency control technique [5] to handle synchronization of generic BST operations, such as rotations and node link/unlinks. To be self contained, we summarize this technique in Sect. 4.1. Section 4.2 then describes how we incorporate the CBTree into it; due to space limitations, pseudo-code is presented in the extended version of the paper [18]. Section 4.3 describes our *single-adjuster* optimization. Section 4.4 describes the Lazy CBTree, a variant of the CBTree meant to reduce the overhead caused by calculating potential differences during the traversal.

4.1 Optimistic Concurrent BSTs

Bronson et al. implement traversal through the tree without relying on read-write locks, using *hand-over-hand optimistic validation*. This is similar to hand-over-hand locking [19] 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.

4.2 Concurrent CBTree Walk-Through

We represent a node v's weight $W(v)$, equal to the total number of accesses to v and its descendants, with three counters: $selfCnt$, counting the total number of accesses to v, $rightCnt$ for the total number of accesses to items in v's right subtree, and $leftCnt$, 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 the same point in time. 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, notice 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 Fig. 1's notation, 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 that replaces z in the tree. 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 [20] of the CBTree therefore follows from the linearizability of Bronson et al.'s algorithm.

Rotations: To do a rotation, the CBTree first acquires locks on the nodes whose links are about to change in parent-to-child order: z's parent, z, y, and for a double rotation also x (using Fig. 1's notation). It then validates that the relationship between the nodes did not change and performs the rotation which is done exactly as in Bronson et al.'s algorithm, changing node version numbers as required and so on. After the rotation the appropriate counters are updated to reflect the number of accesses to the new subtrees.

Counter Maintenance: Maintaining consistent counters requires synchronizing with concurrent traversals and rotations. While traversals can synchronize

by atomically incrementing the counters using `compare-and-swap`, this does not solve the coordination problem between rotations and traversals, and any additional mechanism to synchronize them would be pure overhead because rotations are rare. We therefore choose to 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 at the end of the operation (i.e., not during the intermediate retries that may occur). In the extended version [18] we discuss the races that can occur with this approach and show that such races – if they occur – do not keep the CBTree from obtaining short access paths for frequent items.

Operation Implementation: A `lookup` is a traversal. Insertion is a traversal that terminates by adding a new item or updating the current item's value. Our `delete` implementation follows Bronson et al.'s approach [5], marking a node as deleted and unlinking it when it has one or no children.

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 the extended version [18]. 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.

4.3 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. In this case, 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 Sun's UltraSPARC T2 Plus architecture 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.

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, as we did in our evaluation.

4.4 The Lazy CBTree

Calculating potential differences during the CBTree traversal, which involves multiplication and division instructions, has significant cost on an in-order processor such as Sun's UltraSPARC T2. When no rotation is performed – as is usually the case – the calculations are pure overhead. We have therefore developed the Lazy CBTree, a variant of the CBTree that greatly reduces this overhead by not making rotation decisions during a `lookup` traversal.

A Lazy CBTree traversal makes no rotation decisions along the way. When reaching the destination node, the operation makes a single rotation decision which is based only on counter comparisons, and then proceeds to update the counters along the path back to the root. If the operation is an `insert()` of a new item, it may make more (cheap) rotation decisions as it unfolds the path back to the root. We refer the reader to the extended version for details [18]. While the CBTree analysis does not apply to the Lazy CBTree, in practice the Lazy CBTree obtains comparable path lengths to CBTree but with much lower cost per node traversal, and so obtains better overall throughput.

5 Experimental Evaluation

In this section we compare the CBTree's performance to that of the splay tree, treap [9] and AVL algorithms. All implementations are based on Bronson et al.'s published source code. Benchmarks are run on 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.

Overall, we consider the following implementations: (1) **CB**, CBTree with decaying of node counters disabled, (2) **LCB**, the lazy CBTree variant (Sect. 4.4), (3) **Splay**, Daniel Sleator's sequential top-down splay tree implementation [21] with a single lock to serialize all operations, (4) **Treap**, and (5) **AVL**, Bronson et al.'s relaxed balance AVL tree [5]. Because our single 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 deletions are done as in **Alg** for all threads. Note that SplayOneAdjuster is implemented using Bronson et al.'s code to allow `lookup`s to run concurrently with the adjuster's rotations.

5.1 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 [22], (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 [23], 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 [24]. 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 `insert`s or `delete`s. 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 going through entire sequence.

Table 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 2 shows the throughput, the number of operations completed by all the threads during the duration of the test.

CBTree obtains the best path length, but this does not translate to the best performance: on the UltraSPARC, while CBTree scales well, its throughput is

Table 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.

	AVL	Splay	Treap	CBTree	Lazy CBTree	Splay	Treap	CBTree	Lazy CBTree
						Single adjuster			
Average path length									
books	17.64	10.63, 12.06	11.19	**9.54**, **8.64**	**9.71**, **9.11**	11.71 11.71	11.43 11.43	**10.13**, **11.06**	**10.30**, **10.68**
isp	17.86	9.09, 12.89	14.64	**11.13**	**11.95**, **11.38**	13.39	14.46, 15.1	**12.33**, **13.25**	**12.41**, **13.49**
youtube	14.35	8.52, 13.31	15.75	**11.81**	**12.06**, **11.87**	13.47	15.84	**12.27**	**12.24**
Rotations per operation									
books	0	9.16	< 0.01	$<$ **0.01**	**0.02**	2.95	0.09	**0.02**	**0.02**
isp	0	8.09, 10.45	0.03	**0.01**	**0.03**	2.65, 3.10	0.25	**0.01**	**0.03**
youtube	0	7.52, 10.73	< 0.01	$<$ **0.01**	$<$ **0.01**	3.17	0.11	**0.03**	**0.02**

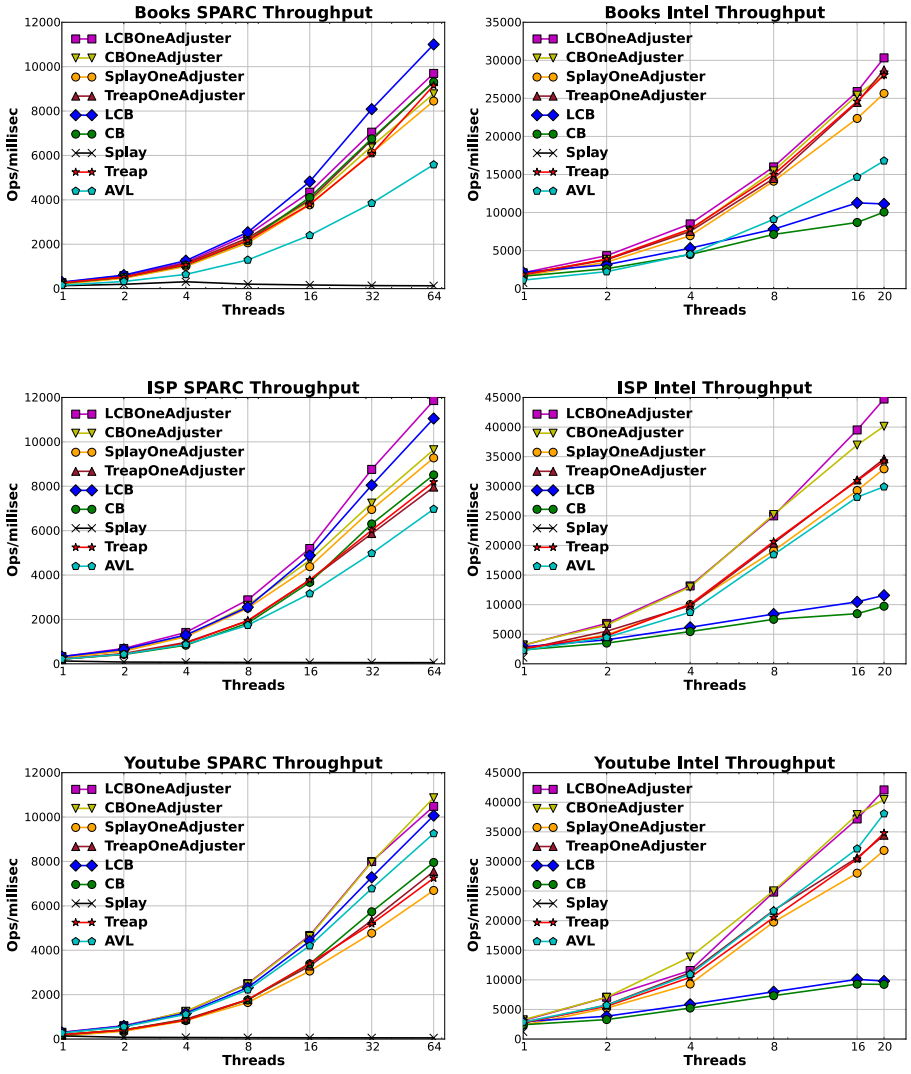

Fig. 2. Test sequence results. **Left:** Sun UltraSPARC T2+ (up to 64 hardware threads). **Right:** Intel Westmere EX (up to 20 hardware threads).

lower than some of the other algorithms due to the cost of calculating potential differences, and on the Intel Xeon E7 CBTree scales poorly because the counter updates serialize the threads (Sect. 4.3). Lazy CBTree, which avoids computing potential differences, outperforms all algorithms except single adjuster variants on the UltraSPARC, but also scales poorly on the Intel.

The single adjuster solves the above problems, making CBOneAdjuster and LCBOneAdjuster the best performers on both architectures. For example, on

the **isp** sequence, CBOneAdjuster and LCBOneAdjuster outperform treap, the next best algorithm, respectively by 15% and 30% on the Intel and by 20% and 50% on the UltraSPARC at maximum concurrency. Because Lazy CBTree on the UltraSPARC incurs little overhead, if LCBOneAdjuster obtains significantly worse path length (e.g., on the **books** sequence), it performs worse than Lazy CBTree. The treap high performance is because an operation usually updates only 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 1.6× at 64 threads on the UltraSPARC machine.

The splay's tree coarse lock prevents it from translating its short path length into actual performance. Applying our single adjuster optimization allows readers to run concurrently and benefit from the adjuster's splaying, yielding a scalable algorithm with significantly higher throughput. Despite obtaining comparable path lengths to CBOneAdjuster, the SplayOneAdjuster does $> 100\times$ more rotations than CBOneAdjuster, which force the concurrent traversals to retry. As a result, CBOneAdjuster outperforms SplayOneAdjuster.

Additional Experiments: The extended version [18] describes additional experiments: (1) evaluating the algorithms on synthetic skewed workloads following a Zipf distribution, (2) examining how the algorithms adjust to changes in the usage pattern, (3) measuring performance under different ratios of `insert`/`delete`/`lookup`, and (4) showing that CBTree is robust to lost counter updates.

Future work. We intend to experiment with other CBTree variants, including the one mentioned at the end of Sect. 3.2, as well as with a top-down version of semi-splaying.

Acknowledgments. This work was supported by the Israel Science Foundation under grant 1386/11, by the Israeli Centers of Research Excellence (I-CORE) program (Center 4/11), and by the BSF. Robert E. Tarjan's work at Princeton is partially supported by NSF grant CCF-0832797. Adam Morrison is supported by an IBM PhD Fellowship.

References

1. Gill, P., Arlitt, M., Li, Z., Mahanti, A.: YouTube traffic characterization: a view from the edge. In: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC 2007, pp. 15–28. ACM, New York (2007)
2. Mahanti, A., Williamson, C., Eager, D.: Traffic analysis of a web proxy caching hierarchy. IEEE Network 14(3), 16–23 (2000)
3. Cherkasova, L., Gupta, M.: Analysis of enterprise media server workloads: access patterns, locality, content evolution, and rates of change. IEEE/ACM Transactions on Networking 12(5), 781–794 (2004)
4. Sripanidkulchai, K., Maggs, B., Zhang, H.: An analysis of live streaming workloads on the internet. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, IMC 2004, pp. 41–54. ACM, New York (2004)

5. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 257–268. ACM, New York (2010)
6. Hanke, S., Ottmann, T., Soisalon-soininen, E.: Relaxed Balanced Red-black Trees. In: Bongiovanni, G., Bovet, D.P., Di Battista, G. (eds.) CIAC 1997. LNCS, vol. 1203, pp. 193–204. Springer, Heidelberg (1997)
7. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the ACM 32, 652–686 (1985)
8. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City
9. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica 16, 464–497 (1996), doi:10.1007/s004539900061
10. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. In: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, STOC 1972, pp. 137–142. ACM, New York (1972)
11. Bent, S.W., Sleator, D.D., Tarjan, R.E.: Biased 2-3 trees. In: Proceedings of the 21st Annual Symposium on Foundations of Computer Science, FOCS 1980, pp. 248–254. IEEE Computer Society, Washington, DC (1980)
12. Feigenbaum, J., Tarjan, R.E.: Two new kinds of biased search trees. Bell System Technical Journal 62, 3139–3158 (1983)
13. Baer, J.L.: Weight-balanced trees. In: American Federation of Information Processing Societies: 1975 National Computer Conference, AFIPS 1975, pp. 467–472. ACM, New York (1975)
14. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2010, pp. 131–140. ACM, New York (2010)
15. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, pp. 161–170. ACM, New York (2012)
16. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM 33, 668–676 (1990)
17. Tarjan, R.E.: Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods 6(2), 306–318 (1985)
18. Afek, Y., Kaplan, H., Korenfeld, B., Morrison, A., Tarjan, R.E.: CBTree: A practical concurrent self-adjusting search tree. Technical report (2012)
19. Bayer, R., Schkolnick, M.: Concurrency of operations on b-trees. In: Readings in Database Systems, pp. 129–139. Morgan Kaufmann Publishers Inc., San Francisco (1988)
20. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 12, 463–492 (1990)
21. Sleator, D.D.: Splay tree implementation, `http://www.link.cs.cmu.edu/splay`
22. Project Gutenberg, `http://www.gutenberg.org/`
23. kc claffy, Andersen, D., Hick, P.: The CAIDA anonymized 2011 internet traces, `http://www.caida.org/data/passive/passive_2011_dataset.xml`
24. Zink, M., Suh, K., Gu, Y., Kurose, J.: Watch global, cache local: YouTube network traffic at a campus network - measurements and implications. In: Proceeding of the 15th SPIE/ACM Multimedia Computing and Networking Conference, vol. 6818 (2008)