

Concurrent Search Tree by Lazy Splaying

Yehuda Afek Boris Korenfeld Adam Morrison

School of Computer Science
Tel Aviv University

Abstract

In many search tree (maps) applications the distribution of items accesses is non-uniform, with some *popular* items accessed more frequently than others. Traditional *self-adjusting* tree algorithms adapt to the access pattern, but are not suitable for a concurrent setting since they constantly move items to the tree's root, turning the root into a sequential hot spot. Here we present *lazy splaying*, a new search tree algorithm that moves frequently accessed items close to the root without making the root a bottleneck. Lazy splaying is fast and highly scalable making at most one local adjustment to the tree on each access. It can be combined with other sequential or concurrent search tree algorithms. In the experimental evaluation we integrated lazy splaying into Bronson et. al.'s optimistic search tree implementation to get a concurrent highly scalable search tree, and show that it significantly improves performance on realistic non-uniform access patterns, while only slightly degrading performance for uniform accesses.

Contact Author:

Yehuda Afek

Phone: +972-544-797322

Fax: +972-3-640-9357

Email: afek@tau.ac.il

Post: School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel

1 Introduction

It is common knowledge, in sequential computing, that although self-adjusting binary search trees (BST), e.g., Splay tree, are theoretically better than balanced BST, e.g., AVL and red-black tree, specially on skewed access sequences, in practice red-black and AVL trees outperform even on skewed sequences [4] (except for extremely skewed sequences). This is mostly because of the self-adjusting overheads, i.e., the constant is too big. Here we show that in concurrent multi-core processing it is *not* the case. We present a new self-adjusting concurrent BST, called Lazy Splaying, that outperforms other known non-adjusting (but balanced, like AVL) BSTs on skewed access sequences. This is a surprising finding since the known self-adjusting BST (different variants of Splay Tree) all modify the root with each access, thus making it an immediate sequential bottleneck. Our new technique overcomes this bottleneck by splaying the tree in a lazy manner which ends up being much more efficient. Our experimental results show that it is also more efficient in the single thread case, on skewed sequences.

Binary Search Tree (BST) is a common and popular data structure for storing an ordered set of items that supports in addition to the standard *insert*, *lookup*, and *remove* operations, *range* operations and *predecessor/successor* operations. Keeping the tree balanced guarantees that the cost of each operation is bounded by $O(\log n)$, where n is the number of items in the tree, and Balanced BSTs, such as, AVL [2] and red-black trees [3] are popular data structures in many software systems, and data-bases. The average and worst case access time in these data structures if each item has the same (uniform) probability of being accessed is logarithmic in the tree size.

However, in practice most access sequences are non-uniform, for instance 80% of the accesses are to 20% of the items [6, 7, 9]. In this situation Optimum tree algorithms [11–13] build trees which provide optimal average access time, but they need the frequencies be fixed and known prior to the tree construction. Of course this is not realistic for online algorithms and can be used only in static trees.

Biased search tries [5, 10] provide optimal with constant factor average access time along with support of modifications in the tree. However, they also requires that the frequencies of the items will be known and a change in the estimated frequency requires costly operations.

Self-adjusting BSTs have been suggested in which frequently accessed items are automatically migrated towards the root of the tree, thus achieving much better average access time over a sequence of accesses, without affecting the worst case (assuming they are kept balanced).

The most famous self-adjusting BST is the splay tree [15] which get the frequently accessed items at the top of the tree by moving each accessed node to the root of the tree while splaying the tree by a sequence of weighted local re-balancing operations on the path from the accessed node to the root. This way frequently accessed nodes end up closer to the root and are found faster when searched again. The major drawback of the splay tree is that each access (insert, look-up or remove) to an item not at the root results in a sequence of tree rotations up to the root. This drawback has two consequences: in sequential computing each operation has a large constant overhead which usually outweighs the logarithmic gain of a binary tree. In concurrent computing the situation is even worse, since almost all operations modify the root, the root becomes a hot sequential bottleneck that makes the structure non-scalable at best. Due to these, despite its good theoretical properties

splay tree is not widely used in practice.

In this paper we propose a new splaying technique, called Lazy Splaying, that overcomes both drawbacks and experimentally wins in both sequential and concurrent environments with skewed data sequences, such as Zipf with skew parameter greater than 0.8 (in which 80% of the accesses is to 37% of the items). In uniform access sequences lazy splaying is still competitive though slightly worse than AVL and red-black tree. The key idea, is to do at most one local tree rotation (re-balancing) per access as a function of historical frequencies. This way an item accessed very frequently does full splaying but over few accesses and ends up high in the tree, on the other hand infrequently accessed items will not get enough pushes up the tree and will remain at the bottom part of the tree. Moreover, the decision on whether to do one local tree rotation or not, depends on the past frequencies of the item and its neighboring subtrees. Like in other online algorithms the history serves as a good estimate on future accesses. Unlike in splay tree, in the steady state with fixed access frequencies there should be almost no re-balancing at all, and most of the splaying overhead is thus removed, making it a good competitor to AVL and other BST specially with skewed access sequences.

Moreover, lazy splaying also resolves the standard splaying root-bottleneck, making it a highly concurrent with good scalability BST. The reason is that it makes at most one rotation per access and does not continue with lock based modifications up to the root. That is, we distinguish between two types of updates during an operation, tree rotations and modifications that have to be done with locking and in a consistent way, and counting the number of operations in different parts of the tree which can be done asynchronously without locking. As we show the algorithm is robust to synchronization hazards on these counters. As far as we know it is the first concurrent scalable self adjusting algorithm. Our extensive experimental testing shows that lazy splaying performs better than top of the art algorithms on skewed access patterns, and the performance advantage grows as the skewness parameter increases (getting farther away from uniform). Moreover, lazy splaying is cache friendly. Since most of the operations are at the top part of the tree, the top part may fit into the cache.

From an implementation point of view, our re-balancing technique can be integrated into most single thread or concurrent BST implementations by replacing their re-balancing method with lazy splaying. In our testing we have integrated lazy splaying with the efficient concurrent implementation of balanced AVL tree provided by Bronson, Casper, Chafi, and Olukotun in [8].

2 Lazy Splaying

A Binary Search Tree (BST) is a binary tree based data structure containing a set of items each with a value from a totally ordered domain. Below we might interchange item for the value associated with the item. In addition to the value each item may contain other data related to the item which is beyond our concern. Each node o in the tree holds one item with value $o.v$ such that, the left subtree of o holds items smaller than $o.v$ and the right subtree holds items larger than $o.v$. Any operation on an item starts by searching the item in the BST (even insert to make sure the item is not there already). A search for an item starts going down from the root, going left and right at each node o if the searched item is smaller or larger than the item at o , respectively, until reaching a node containing the item, or an empty subtree indicating the item is not in the tree. In such a

case the item can be inserted at the point where the search stopped.

Following splay tree BST techniques [15] in lazy splaying we perform one tree rotation (kind of re-balancing) at the node containing the item we operate on, moving subtrees with more frequently accessed nodes one level up on the account of moving less frequently accessed subtree(s) one level down. Specifically we perform conditional *zig-zag* or *zig* operations as they are named in [15], and as explained below. Notice that Lazy splaying is substantially different than the semi-splaying variations suggested in Section 5 of [15], where splaying is also conditioned but on different parameters and when done, it is still done all the way to the root (sometimes only on a subset of the nodes on the path to the root).

In lazy splaying in addition to the item's value each node o has three counters, *selfCnt* which is an estimate on the total number of operations that has been performed on the item in o (number of $\text{find}(o.v)$ and $\text{insert}(o.v)$ operations), *rightCnt* and *leftCnt* which are an estimate on the total number of operations that have been performed on items in the right and left sub-tree respectively. Each $\text{find}(i)$ and $\text{insert}(i)$ operation increments *selfCnt* of the node containing i . When node i is found in the tree all the nodes along the path from the root to i 's parent increase their *rightCnt*/*leftCnt* counter depending on whether i was in their right or left sub-tree. Actual *insert* and *remove* operations are performed concurrently exactly as in Bronson et.al. [8]. We simply replaced the re-balance function in their code with our new lazy splaying re-balancing code.

The re-balancing (tree rotation) is performed just before incrementing the counter *selfCnt*. See Figure 1 and Listing 4 for the code and description of *zig* and *zig-zag*. *Zig-zag* is carried out if the total number of accesses to the node right subtree is larger than the total number of accesses to the node-parent and its right subtree. If *zig-zag* was not performed then *zig* is performed if the total number of accesses to the node and its left subtree is larger than the total number of accesses to the node-parent and its right subtree. After the rotation *rightCnt* and *leftCnt* counters are updated to represent the number of accesses in the new right/left sub-tree respectively. Notice that *zig* and *zig-zag* have a symmetric mirror operations when the subtree at node p leans to the right.

To avoid a chain of nodes where all nodes are left (or all nodes are right) parents of their children in the case of descending / ascending insertion order, when a new node inserted to the tree re-balancing operation performed from this new node up to the root.

Lazy splaying re-balancing can be integrated with different multi-thread binary search tree implementations by replacing the re-balancing code with ours. Here we integrated lazy splaying with the implementation of practical AVL binary search tree [8] and replaced the AVL re-balancing according to height. In [8] the function `fixHeightAndRebalance` is replaced with our re-balancing code. One advantage of doing it like this, is that any difference in the experimental performances when comparing the two implementations can be attributed to the new re-balancing code. Hence the locking and the consistency checks are done as in [8]. The locks are taken in order from the parent to child. If thread A does a *zig* rotation it doesn't need to lock `node.right`. If `node.right` is changed concurrently by another thread B then `node.right` is the grand parent in B 's rotation. Otherwise B must lock the node which is locked by A . Since only `node.right` parent field is changed and it isn't changed by B , operations are atomic. After taking a lock, the algorithm performs a check to assure the node is not modified by a concurrent thread. If the node has been modified by a concurrent thread then the algorithm unlocks its locks and gives up the rotation.

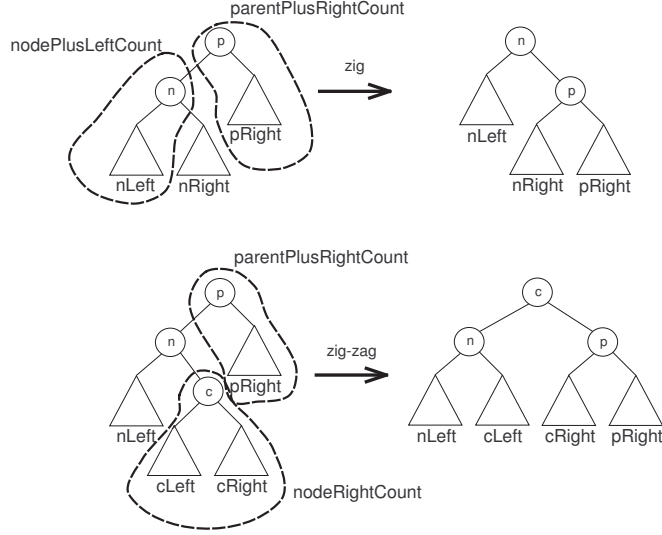


Figure 1: Splaying steps. Zig: when the number of operations on n and on nodes in $nLeft$ is larger than the number of operations on p and on nodes in $pRight$. Zig-zag: when the number of operations on c and on nodes in $cLeft$ and $cRight$ is larger than the number of operations on p and on nodes in $pRight$.

Counters Estimation: While rotations (which are all the possible structural modifications of the tree) are protected by locks, the different counter increments at the nodes are not protected by locks and may suffer from race hazards. These potential hazards have no effect on the safety properties of the operations and may only effect the correctness of the counter values, which in turn may effect performances. However, as we show and extensively examined experimentally these inaccuracies have negligible effect on the performances (see Figure 9 for testing with fake large hazards), while the elimination of the extra locking operations (CAS operations) significantly improves the overall performances.

Long splaying: Since the counters are updated in an asynchronous un-safe manner we add a safety belt watch-dog check in the code, that if a node at depth larger than $2 \log n$ is reached then full semi-splaying (following semi-splaying of [15]) along the path from the node to the root is performed. We keep this check, although rarely reached, as a measure of protection and guarantee on worst case per operation performance. Furthermore, the inclusion of this semi-splaying along the path enables us to rely on Theorem 7 in [15] and to claim worst case amortized complexity of $O(\log n)$. This check made us add code that estimates $\log n$ to within $+1$ or -1 from the true value. Essentially each thread holds a local count of the total number of new items it has inserted minus the number of items it has removed from the tree, and whenever that count either doubles or halves it reads the counter of all the threads and update the estimate accordingly. If it fails to update the new estimate due to a conflict (done with *CAS*) it re-scans the new total value and tries to *CAS* again.

Listing 1: AttemptInsert function

```

1 attemptInsert(key, parent, node, height)
2 {
3     if (key == node.key) {
4         if (height >= ((2 * log_size))) {
5             SemiSplay(node);
6         } else {
7             Rebalance(parent, node);
8         }
9         node.selfCnt++;
10        return node.value;
11    }
12    while (true) {
13        child = node.child(key);           //child in the direction to the key
14
15        //check if node is not changed or removed by another thread
16        if ( isInvalid (node) ) {
17            return SpecialRetry;
18        }
19        if ( child == null ) {
20            child = NewNode(key, node);           //create a new node and link it to node
21            if (height >= ((2 * log_size))) {
22                SemiSplay(child);
23            }
24            return null;           //not found!
25        } else {
26            result = attemptInsert(key, node, child, height+1);
27            if (result != SpecialRetry) {
28                if (direction to child == Left){
29                    node.leftCnt++;
30                } else {
31                    node.rightCnt++;
32                }
33                if (result == null){
34                    //for new node check if re-balancing needed
35                    //Find the current child of the parent since it may changed due to rotation in recursive call.
36                    curr_node = parent.child(key);
37                    //child in the direction to the key
38                    Rebalance(parent, curr_node);
39                }
40                return result;
41            } // else RETRY
42        }
43    }
44 }
45 \end{Code}
46
47 \begin{Code}
48 \centering
49 \lstset {caption=AttemptGet function,label=get, captionpos=t, frame=single}
50 \begin{lstlisting} [language=java,
51 emph={parent, node, selfCnt, leftCnt, rightCnt, right, left },emphstyle=\color{darkBlue},tabsize=2]
52 attemptGet(key, parent, node, height)
53 {
54     if (key == node.key) {
55         if (height >= ((2 * log_size))) {
56             SemiSplay(node);
57         } else {
58             Rebalance(parent, node);
59         }
60         node.selfCnt++;
61         return node.value;
62     }
63     while (true) {
64         child = node.child(key);           //child in the direction to the key
65
66         //check if node is not changed or removed by another thread
67         if ( isInvalid (node) ) {
68             return SpecialRetry;
69         }
70         if ( child == null ) {
71             if (height >= ((2 * log_size))) { 5
72                 SemiSplay(node);
73             }
74             return null;           //not found!
75         } else {
76             result = attemptGet(key, node, child, height+1);
77             if (result != SpecialRetry) {
78                 if (direction to child == Left){
79                     node.leftCnt++;
80                 } else {
81                     node.rightCnt++;
82                 }
83                 if (result == null){
84                     //for new node check if re-balancing needed
85                     //Find the current child of the parent since it may changed due to rotation in recursive call.
86                     curr_node = parent.child(key);
87                     //child in the direction to the key
88                     Rebalance(parent, curr_node);
89                 }
90                 return result;
91             } // else RETRY
92         }
93     }
94 }
95 \end{Code}

```

Listing 2: Insert function

```

1 insert(key, id) {
2     if (attemptInsert(key, null, root, 0) == null){
3         //item was not found and was inserted
4         localCounter[id]++;
5         if (localCounter[id]>max[id]){
6             max[id]=localCounter[id];
7         }
8         if (localCounter[id]>2*min[id]){           //local counter was doubled
9             //start remembering max and min from now on.
10            max[id]=min[id]=localCounter[id];
11            succeeded = false;
12            while (!succeeded){
13                prevLogSize = logSize;
14                prevVersion = version;
15                //scan all counters and update log_size
16                totalSize = 0;
17                for (i=0;i<NUM_OF_THREADS;i++){
18                    totalSize +=localCounter[i];
19                }
20                if (log( totalSize ) != logSize){
21                    succeeded = compareAndSet(prevLogSize, prevVersion,
22                                                log( totalSize ), prevVersion+1);
23                }else{
24                    succeeded=true;
25                }
26            }}}
27 /* highTreshold and lowTreshold values in border cases are omitted. */

```

Listing 3: Remove function

```

1 remove(key, id) {
2     if (attemptRemove(key, null, root, 0) != null) {
3         //item was found and removed (at least logically)
4         localCounter[id]--;
5         if (localCounter[id]<min[id]){
6             min[id]=localCounter[id];
7         }
8         if (localCounter[id]<max[id]/2){           //local counter was doubled
9             //start remembering max and min from now on.
10            max[id]=min[id]=localCounter[id];
11            succeeded = false;
12            while (!succeeded){
13                prevLogSize = logSize;
14                prevVersion = version;
15                //scan all counters and update log_size
16                totalSize = 0;
17                for (i=0;i<NUM_OF_THREADS;i++){
18                    totalSize +=localCounter[i];
19                }
20                if (log( totalSize ) != logSize){
21                    succeeded = compareAndSet(prevLogSize, prevVersion,
22                                                log( totalSize ), prevVersion+1);
23                }else{
24                    succeeded=true;
25                }
26            }
27 }
28 }
29 }
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }
46 }
47 }
48 }
49 }
50 }
51 }
52 }
53 }
54 }
55 }
56 }
57 }
58 }
59 }
60 }
61 }
62 }
63 }
64 }
65 }
66 }
67 }
68 }
69 }
70 }
71 }
72 }
73 }
74 }
75 }
76 }
77 }
78 }
79 }
80 }
81 }
82 }
83 }
84 }
85 }
86 }
87 }
88 }
89 }
90 }
91 }
92 }
93 }
94 }
95 }
96 }
97 }
98 }
99 }
100 }
101 }
102 }
103 }
104 }
105 }
106 }
107 }
108 }
109 }
110 }
111 }
112 }
113 }
114 }
115 }
116 }
117 }
118 }
119 }
120 }
121 }
122 }
123 }
124 }
125 }
126 }
127 }
128 }
129 }
130 }
131 }
132 }
133 }
134 }
135 }
136 }
137 }
138 }
139 }
140 }
141 }
142 }
143 }
144 }
145 }
146 }
147 }
148 }
149 }
150 }
151 }
152 }
153 }
154 }
155 }
156 }
157 }
158 }
159 }
160 }
161 }
162 }
163 }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }
172 }
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }
182 }
183 }
184 }
185 }
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
```

Listing 4: Rebalance function

```

1 Rebalance(Node parent, Node node){
2
3     nodePlusLeftCount = node.selfCnt + node.leftCnt;
4     parentPlusRightCount = parent.selfCnt + parent.rightCnt;
5     nodeRightCount = node.rightCnt;
6
7     //decide whether to perform zig-zag step
8     if (nodeRightCount >= parentPlusRightCount){
9         Node grand = parent.parent;
10        synchronized (grand) {
11            if ((grand.left == parent) || (grand.right == parent)) {
12                synchronized (parent) {
13                    if (parent.left == node) {
14                        synchronized (node) {
15                            Node rightChild = node.right;
16                            if (rightChild != null){
17                                synchronized (rightChild){
18                                    ZigZag(grand, parent, node, rightChild);
19                                    parent.leftCnt = rightChild.rightCnt;
20                                    node.rightCnt = rightChild.leftCnt;
21                                    rightChild.rightCnt += parentPlusRightCount;
22                                    rightChild.leftCnt += nodePlusLeftCount;
23                                }
24                            }
25                        }
26                    }
27                }
28            }
29        }
30        //decide whether to perform zig step
31        }else if (nodePlusLeftCount > parentPlusRightCount) {
32            Node grand = parent.parent;
33            synchronized (grand) {
34                if ((grand.left == parent) || (grand.right == parent)) {
35                    synchronized (parent) {
36                        if (parent.left == node) {
37                            synchronized (node) {
38                                Zig(grand, parent, node, node.right);
39                                parent.leftCnt = node.rightCnt;
40                                node.rightCnt += parentPlusRightCount;
41                            }
42                        }
43                    }
44                }
45            }
46        }
47    }
48 }

```

3 Worse case analysis

Since counters are incremented without synchronization they may have wrong values which may result in long path to a node.

To avoid this situation we do long path splay. In [15] were proposed semi-adjusting search trees. Semi-splaying step is similar to splaying step except that in zig-zig step x is not moving to the top. The rotation done between its parent and grand parent and splaying continues from its parent. See picture. Additionally in [15] was suggested to perform semi-splaying only over long paths, when the depth of the node is at least $c \log(N) + c/2$ for any $c > 2$ where N is the number of nodes in the tree. If we do not perform any rotations in lower depths as proposed in the paper the total splaying time is $O(N \log N)$. See theorem 7 Long Splay Theorem in [15].

Our algorithm counts the depth of the node starting from the root. If the depth is higher than $2 \log N$ it will perform semi-splaying up to the root. Otherwise it will perform at most one rotation as described before.

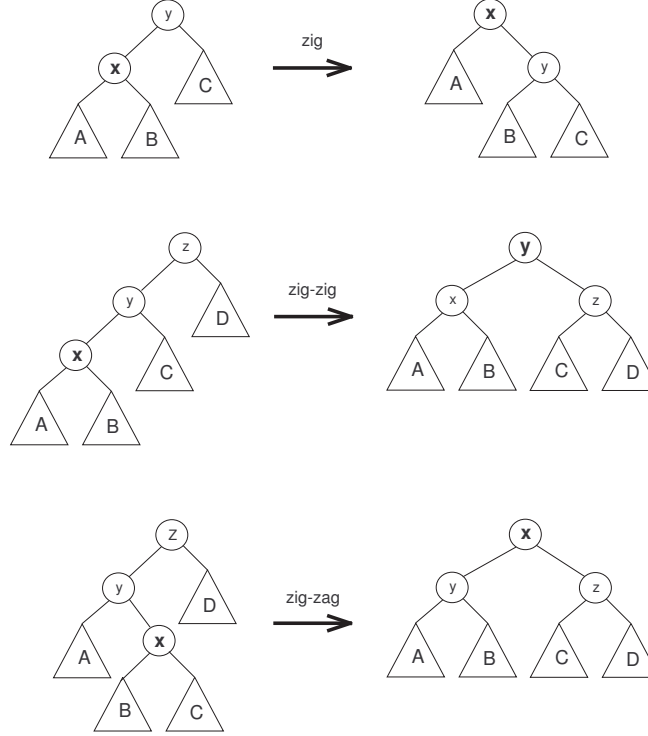


Figure 2: Semi-splaying. Node in bold is the current node of splaying.

Amortized time analysis of the algorithm. Let's $c=4$ be our constant required in Long Splay Theorem. In addition to long paths splaying time of $O(N \log N)$, we perform semi-splaying on shorter paths. These paths are at most $4 \log(N) + 1$ hence the total time to perform them is at most $O(m \log N)$ where m is the total number of operations. Operation on nodes with depth smaller than $2 \log N$, from the same reason have total time of $O(m \log N)$. As in theorem 1 in [15] the decrease in potential over the sequence of rotations algorithm performs over short paths is at most $N \log N$. Summarize all the times above gives that the total time to perform m operations is $O((m + N) \log N)$, hence amortize time of operation is $O(\log N)$.

$\log N$ concurrent calculations. In order to decide whether to perform semi-splaying to the root we should know the $\log N$ value. By $\log N$ it is always considered the highest integer value smaller than $\log N$. Pay attention that for the amortize time analysis we do not need the exact $\log N$ value, since we perform semi-splaying also on shorter paths. Brute force calculation of $\log N$ by using atomic counter for N and calculating $\log N$ from it may become a bottleneck when many threads will concurrently try to update N . Therefore, we use the following algorithm to calculate estimation of $\log N$.

Every processor has a counter that is incremented on every insert of a new item and is decremented once an item is removed. See listing 2 and listing 3. Initially counter value is 0. Therefore the summary of all counters represents the number of nodes in the tree. The thread stores locally the previously calculated global $\log N$ value and summarize the values of all threads counters.

If the log of the summery is higher or lower than the previously calculated global $\log N$, the global

$\log N$ value is updated using *compareAndSet* method. To assure that no other thread changes the value of global $\log N$ and the errors don't accumulate in addition to 8 bit dedicated to $\log N$ value additional 24 bits represent update version. If *compareAndSet* fails due to concurrent update made by another thread and the new value $\log N$ still not equals to calculated by thread i value, thread i will retry the procedure.

Remark: Only 8 bits are used for $\log N$ value since 2^{255} is higher than any practically needed value. Other bits in the atomic integer are used to store the version of the log. This makes the scanning and log calculation an atomic operation and avoids ABA problem.

If the real $\log N$ value increases/decreases by more than one it means that the summary of the counters is at least doubled/halved. This means that at least one local counter was doubled/halved thus the scan was performed and a new $\log N$ was calculated.

Because of the log version, in the time interval between thread i read the previously calculated global $\log N$ and successful call to *compareAndSet* no other thread issues successful *compareAndSet*. Let's suppose that thread i wants to increment global $\log N$ value (the case of decrement is symmetrical). Thread i read other thread j counter value c_j . Since no other thread changed $\log N$ value, it means that all other threads current value c'_j remains $c_j \geq 2 \cdot c'_j$. This means that the value of every counter may be reduced to at most half and so is the summary of all counters. Therefore the error in $\log N$ is at most 1.

Our algorithm performs splaying when the depth is at least $2 \log \tilde{N}$, where \tilde{N} is an estimated N . $2 \log \tilde{N} \leq 4 \log N + 2$ Therefore on every long path we still perform semi-splaying and the amortize time bounds are still hold.

3.1 Adaptiveness to changes in access frequencies

While counting the number of operations on items in different parts of the tree significantly improves the performance on skewed access sequences, it may be problematic in the case that a very popular item becomes in-popular but is still stuck at the top of the tree for a long time. Frequent nodes from the new distribution need a lot of time until they reach high enough counters to beat nodes that lost their popularity. To overcome this we add an exponential decay function to the counters. 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. Every *clockCycle* a *globalClock* is incremented by 1. Before comparing with any counter, if *timeStamp* of the corresponding node is smaller than the *globalClock* the following *updateCounters* function is called. This function divides by 2 each of the nodes' counters for each *clockCycle* time that has elapsed since the last time the counters have been updated. The effect of the adaptive mechanism is tested and compared with a non-adaptive version in Figure 8.

Listing 5: UpdateCounters function

```

1  updateCounters(Node node) {
2      synchronized (node) {
3          // check that not modified by concurrent operation
4          if (node.timeStamp < globalClock) {
5              int timeDif = globalClock - node.timeStamp;
6              node.selfCnt = node.selfCnt >> timeDif;          /* shift right timeDif times */
7              node.rightCnt = node.rightCnt >> timeDif;
8              node.leftCnt = node.leftCnt >> timeDif;
9              node.timeStamp = globalClock;
10         }}}

```

To avoid race condition in which several threads divide a node's counter by two simultaneously we protect the code with a lock (java synchronized block). Over time nodes which were popular and then became unpopular decrease their counters, and move down the tree enabling nodes that become now popular to climb up the tree.

4 Performance

Experimental setup. The experiments run on UltraSPARC T2 Plus server [1] with two Niagara II chips, a multithreading (CMT) processor; each chip has 8 1.165 GHz in-order cores with 8 hardware threads each, for a total of 64 hardware threads. All algorithms are implemented in Java, using the HotSpot Server JVM build 19.0-b09 with Java Runtime Environment version 1.6.0-23-ea. Unless stated otherwise, reported results are averages of three 10-second runs on an otherwise idle machine. Initially the tree is warmed up by a single thread inserting a total of 5 times the key range values into the tree (i.e., for keys in the range 130,000 it inserts 650,000 keys), after which the measurement start.

The following four algorithms are tested and compared:

OptTree - Bronson et. al. [8] implementation, without the extensions which worsen the performance.

UnbalancedTree - *OptTree* in which the AVL re-balancing code is disabled. The trees created by this algorithm are not balanced, hence it is named *UnbalancedTree*. This serves a yard stick to measure how much each of the re-balancing techniques (AVL ala Bronson et. al. or lazy splaying) improves the performances of the implementation.

LazySplayTree - *UnbalancedTree* with lazy splaying re-balancing. The difference between *LazySplayTree* and *UnbalancedTree* is due to the lazy splaying re-balancing method.

LazyAdaptSplayTree - The adaptive version of *LazySplayTree*. Unless stated otherwise *clockCycle* value in all tests is 1 sec.

In the first few tests the values selected for a lookup are Zipf distributed with skew parameter 0.94 in which 80% of the accesses are to 20% of the items as in 80-20 Pareto rule [14]. While the

removes in the benchmark are distributed uniformly, the inserts values are taken from the same Zipf distribution as the lookups. This results in faster return of frequent nodes after they have been removed.

Figure 3 (a) shows the average paths length to accessed nodes in difference size trees. Since the lazy splay tree has shorter average paths length it also provides better performance as can be seen in figure 3 (b). The test were performed with 64 threads on trees with different sizes. Operation ratio was 9% inserts, 1% removes and 90% lookups.

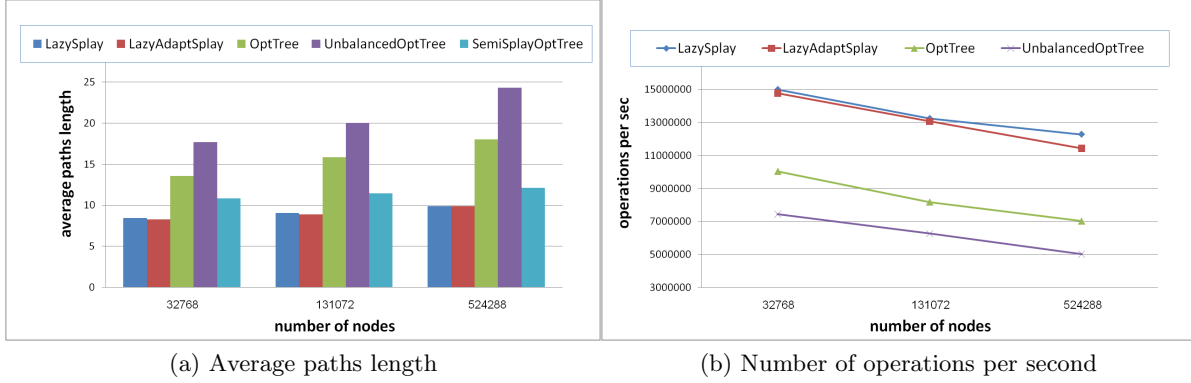


Figure 3: Performance of 64 threads running on trees with different sizes. Operation ratio is 9% inserts, 1% removes and 90% lookups. Zipf exponent (skew parameter) is 0.94 resulting in 80% of accesses to 20% of the items. (a) Average paths length over the access sequence as measured by one out of the 64 threads. (b) Number of operations per second.

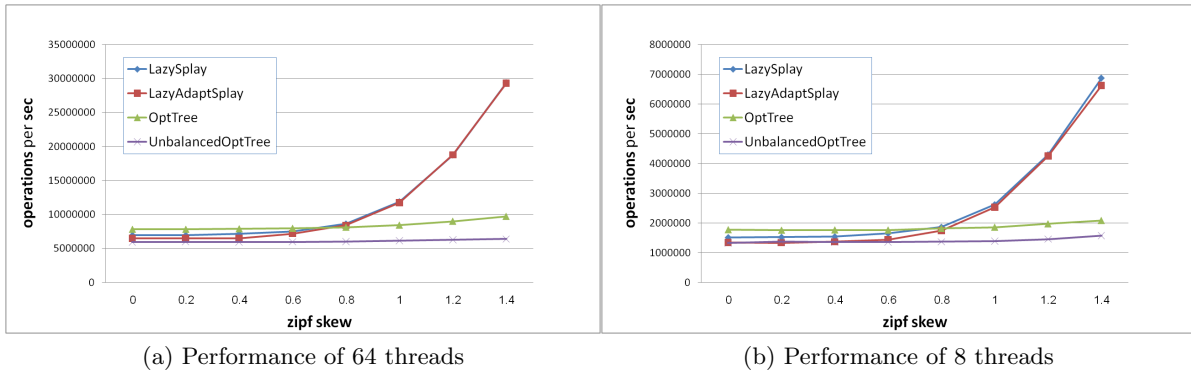


Figure 4: Performance on a tree with 130K nodes as a function of the Zipf exponent. Operation ratio is 9% insert, 1% remove and 90% lookup. (a) Number of operations per second performed by 64 threads. (b) Number of operations per second performed by 8 threads. In skew of 0.8, equilibrium point of LazySplayTree and OptTree performances, 80% of accesses are made to 37% of the items.

Figure 4 compares the performances with different Zipf skew parameters on a tree with 130K nodes. As can be seen in access distributions with zipf exponents larger than 0.8 (80% accesses to 37% of the items) *LazySplayTree* outperforms *OptTree*. The explanation for this can be seen in figure 5. While in *OptTree* average path length remains almost the same for all skew value, in *LazySplayTree* average path length becomes shorter as skew grows.

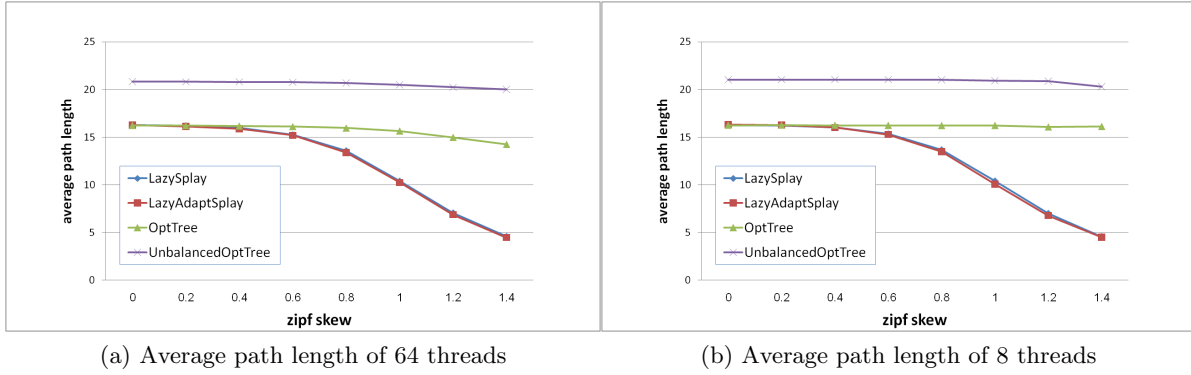


Figure 5: Average path length on a tree with 130K nodes as a function of the Zipf exponent. Operation ratio is 9% insert, 1% remove and 90% lookup. (a) Average path length performed by 64 threads. (b) Average path length performed by 8 threads.

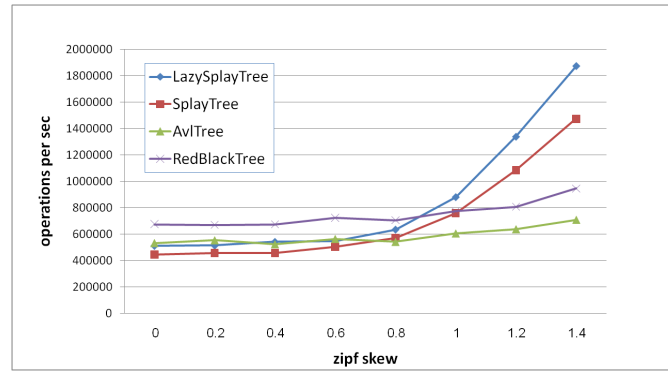


Figure 6: Single thread performance on a 130K nodes tree as a function of the Zipf exponent. Operation ratio is 10% insert and 90% lookup. From skew level of 0.6 and up LazySplayTree outperforms AvlTree, where 80% of the accesses are made to 55% of the items. Until skew level 0.9 RedBlackTree outperforms LazySplayTree, where 80% of accesses are made to 24% of the items.

Figure 6 compares the performances of different Binary Search Tree sequential (single thread) implementations. The following BSTs were tested:

LazySplayTree a single thread implementation that doesn't use locks, or checks and other code needed for synchronization.

SplayTree is an implementation of top-down splay tree by Daniel D. Sleator available from <http://www.link.cs.cmu.edu/splay/>.

AvlTree and **RedBlackTree** are AVL tree and Red Black tree implementations by Mark Allen Weiss available from <http://users.cis.fiu.edu/~weiss/dsajava/code/DataStructures/>.

In Zipf access distributions with exponent smaller than 0.9 (around 80%–40% ratio) **RedBlackTree** outperforms all other algorithms, while with higher exponent values **LazySplayTree** outperforms all others with an increasing gap. The small increase in performances by **AvlTree** and **RedBlackTree** with high exponent values is because of the better caching while they perform many operations on

the same paths to frequent nodes. Since the *AvlTree* and *RedBlackTree* implementations that we use are missing remove implementation we compared *LazySplayTree* only against *SplayTree* when operation ratio did not included removes. These tests also show that *LazySplayTree* outperforms *SplayTree* in a single thread implementation.

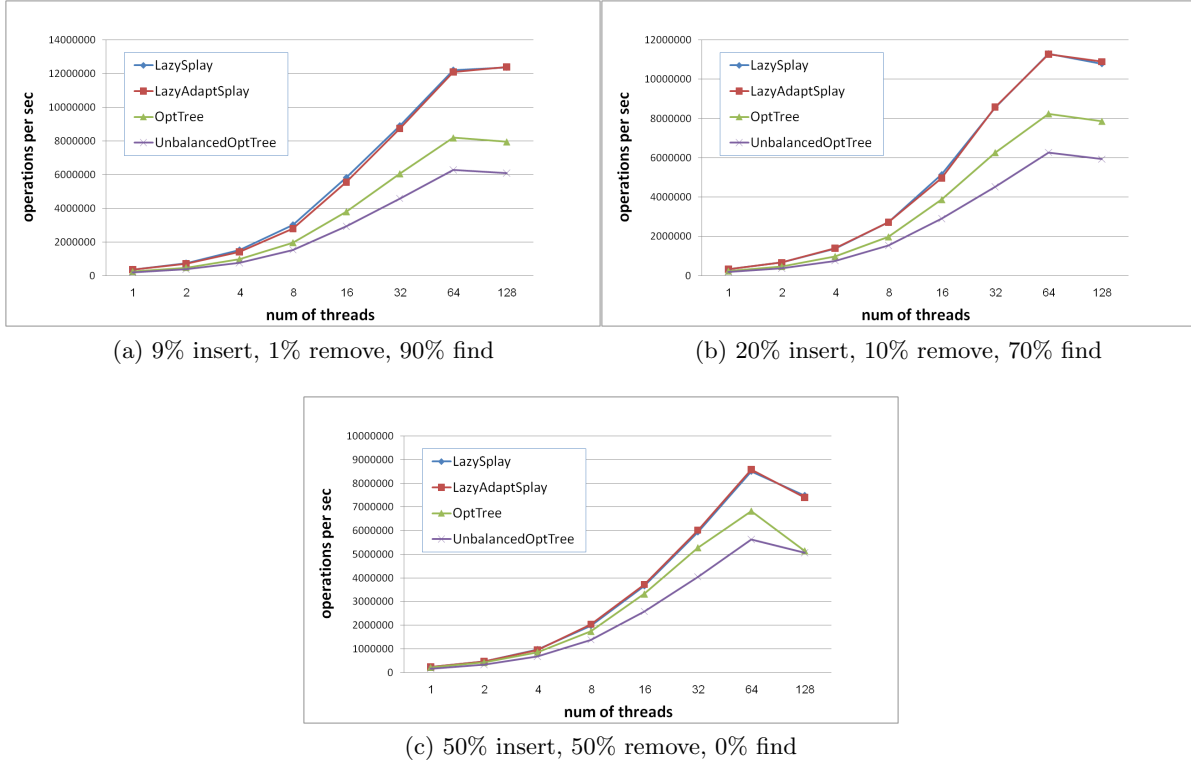


Figure 7: Scalability test: Performance on a 130K nodes tree with different operation mix as a function of the number of threads (maximum of 64 hardware threads, i.e., at 128 there is context switching). Zipf exponent is 0.94 (80% – 20% ratio). Remove operations items are uniformly distributed.

The scalability of the different implementations is shown in Figure 7, where the performance of the algorithms throughput as a function of the number of threads (from 1 to 128) on different operations mix is given. Zipf exponent is 0.94. When the percentage of removes, which are uniformly distributed, grows the performance of *LazySplayTree* decreases. Pay attention that the number of hardware threads in the test is 64. 128 threads results show the effect of context switches on the performance. Even though counters are non-atomic and may be disrupted by context-switches, in practise context-switches on *LazySplayTree* and *LazyAdaptSplayTree* have the same effect as on *OptTree* and *UnbalancedTree*.

The advantage of *LazyAdaptSplayTree* over *LazySplayTree* is exemplified in Figure 8 where we run the following test with 100% lookup operations. On initially filled tree, in the first 15 seconds one thread executes one sequence taken from a Zipf distribution with skew of 0.8 while in the last 15 seconds it executes on a sequence taken from a new sampling of Zipf distribution with the same skew. The *LazyAdaptSplayTree* algorithm is tested with 3 different *clockCycles* values: 1 sec, 100 ms and 10 ms. As can be seen, though in the first 15 seconds *LazySplayTree* and

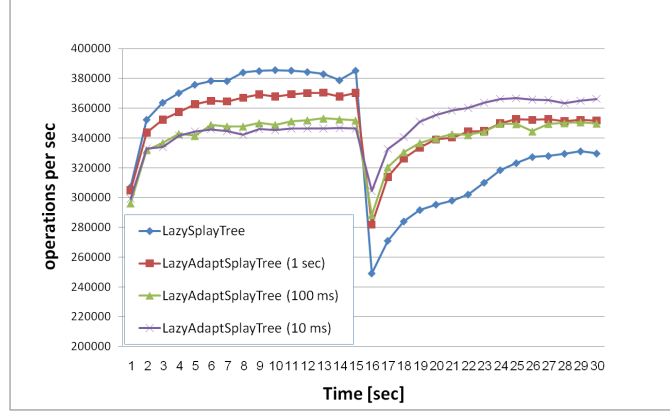


Figure 8: On initially filled tree, first 15 seconds one thread executes 100% lookup operations from one zipf distribution with skew 0.8, while in the last 15 seconds it executes 100% lookup of from a different zipf distribution with the same skew. *LazyAdaptSplayTree* algorithm is tested with 3 *clockCycles* values, 1 sec, 100 ms and 10 ms.

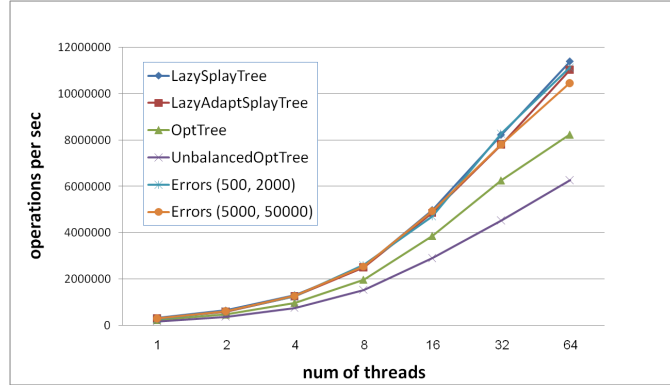


Figure 9: Error injection test. This test simulates the situation in which a scheduler makes context-switches during the increment of the counter. This may result in out dated counter updates, i.e., errors. The Performance on a tree with 130K nodes and different operation mix of 9% insert, 1% remove, 90% lookup. Zipf exponent 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.

LazyAdaptSplayTree with larger *clockCycles* perform better, the drop in performance on a distribution switch is much larger. *LazyAdaptSplayTree* which doesn't divide its counter at all during the test, after distribution switch has previously frequent nodes in the top of the tree while nodes that become now popular are placed in the bottom of the tree. It takes a lot of time until these frequent nodes overcome the previously frequent nodes and take their place in the top part of the tree. *LazyAdaptSplayTree* with small *clockCycle*, on the other hand, has counters with very small values and quickly loses the influence of the previous distribution (short memory). In conclusion, there is a tradeoff between better performance in fixed distributions enabled with larger *clockCycle* values and better adaptiveness to distribution changes with smaller *clockCycle* values.

While the tests in Figure 7 show that context switches do not affect the performance even though the counters are non-atomic, we test the algorithms under harsher conditions. The purpose of the test

is to show the robustness of the algorithm despite unexpected erroneous changes to the counters due to its un protected updates. The results are given in Figure 9: Every thread every *errorFrequency* operations inject an error into one of the node’s counters. *leftCnt* and *rightCnt* were 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 . The tests shown that neither have significant affect on the performances. With *errorValue* < 2000 there is almost no effect. With error -50000 performances are reduced by up to -9.5% . Thus showing that the algorithm is robust to counters inaccuracies.

5 Extensions

Thresholds before making re-balance operations

Since re-balancing is performed when some sum of counters higher than other by only one, there was a concern that many redundant re-balancing operations are performed. We performed a test where re-balancing operation are performed only when the counters of a child are higher than a counters of the parent by some threshold. As showed on figure 10 small threshold has negligible negative effect while higher thresholds have much worse performance.

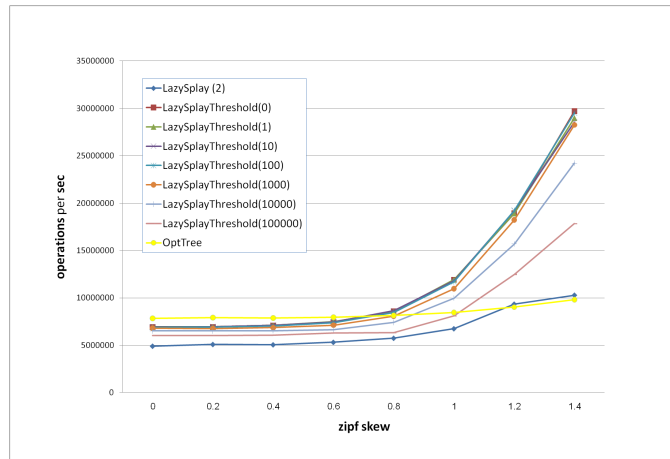


Figure 10: Number of operations per second performed by 64 threads on a tree with 130K nodes as a function of the Zipf exponent. Operation ratio is 9% insert, 1% remove and 90% lookup. *LazySplayThreshold(x)* stands for LazySplay algorithm with threshold parameter x . *LazySplay(2)* is the *LazySplay* algorithm where re-balancing performed only when child counters sum is twice the counter sum of the parent. As can be seen, higher thresholds have worse performance.

Re-balancing up to the root

We checked wether performing re-balancing up to the root can provide a better balanced tree and an improved the performance. *BiasedTree* variation, after accessing the searched node, doesn’t perform the re-balance operation as *LazySplay*. Instead it continues up to the root checking that grand parent half sum of counters is higher than grand child’s sum of counters. If this doesn’t hold, it tries to re-balance as the *LazySplay* does. *BiasedSplayTree* is a combination of *BiasedTree* and *LazySplay*. It performs re-balancing on the accessed node as *LazySplay* does and continues

the checking and the re-balancing up to the root. As seen in figure 11 performing re-balancing up to the root have small negative effect on performance. However not performing the re-balancing on the accessed node greatly reduce the performance.

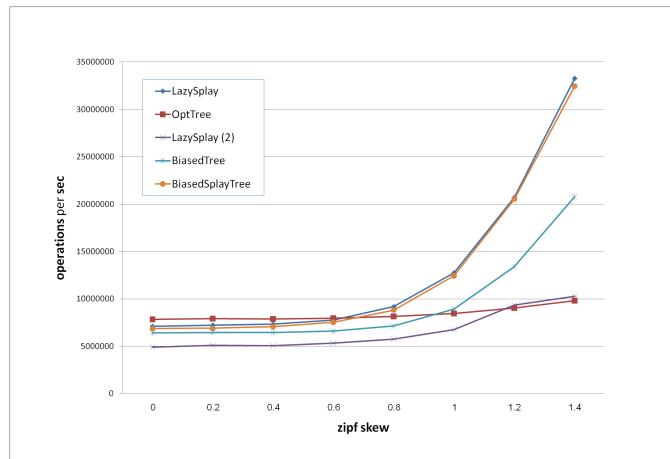


Figure 11: Number of operations per second performed by 64 threads on a tree with 130K nodes as a function of the Zipf exponent. Operation ratio is 9% insert, 1% remove and 90% lookup. *BiasedTree* makes re-balancing only when sum of counters of a grand child higher than half sum of counters of a grand parent, starting from the accessed node up to the root. *BiasedSplayTree* re-balances on accessed node as *LazySplay* does and continues checking up to the root as *BiasedTree* does. As can be seen checking up to the root has small impact on performance. However, not re-balancing on the accessed node greatly reduce the performance.

6 Conclusions

In this paper we have presented the first scalable concurrent self adjusting binary search tree, and a new re-balancing technique - lazy splaying which improves the performance and scalability of self adjusting binary search trees. We believe that self adjusting binary search trees will be no more only a theoretical algorithm, but can be used as a practical single thread and highly scalable multi-thread BST implementation. Future work: 1. it is interesting to see whether the bottleneck created with high zip skew values may be settled. 2. Is there an algorithm with better performance on both uniform and zipf distributions. Possible candidate is a combination between ideas from lazy splaying and other algorithms such as red-black tree or other AVL trees.

References

- [1] OpenSPARC T2 Core Microarchitecture Specification. http://www.opensparc.net/pubs/t2/docs/OpenSPARCT2_Core_Micro_Arch.pdf, 2007.
- [2] G. Adel’son-Vel’ski and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences, volume 145, pages 263-266, 1962. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259–1263, 1962.*
- [3] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

- [4] Jim Bell and Gopal K. Gupta. An evaluation of self-adjusting binary search tree techniques. *Softw., Pract. Exper.*, 23(4):369–382, 1993.
- [5] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased 2-3 trees. In *In Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, pages 248–254. ACM, October 1980.
- [6] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134 vol.1, mar 1999.
- [7] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134 vol.1, mar 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 of Parallel Programming*. ACM, January 2010.
- [9] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29:251–262, August 1999.
- [10] J. Feigenbaum and R. E. Tarjan. Two new kinds of biased search trees. *Bell Syst. Tech. J.*, 62:3139–3158, 1983.
- [11] T.C. Hu and C. Tucker, A. Optimal computer-search trees and variable-length alphabetic codes. *SIAM J. Appl Math.*, 37:246–256, July 1979.
- [12] D.E. Knuth. Optimum binary search trees. *Acta Inf.*, 1:14–25, 1971.
- [13] D.E. Knuth. *The Art of Computer Programming. Vol. 3, Sorting and Searching*. Addison-Wesley Reading, Mass., 1973.
- [14] Vilfredo Pareto. *The Mind and Society [Trattato Di Sociologia Generale]*. Harcourt, Brace., 1935.
- [15] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery*, 32:652–686, July 1985.