Parse Concurrent Structures: BST as an Example

Keren Zhou, Guocheng Niu, Wuzhao Zhang, Xueqi Li, Wenqin Liu

Insititute of Computing Technology, Chinese Academy of Sciences

Abstract

Designing Concurrent data structures' should follow some basic rules. By separating the algorithms into two phases, we present guidelines for scalable data structures, with a analysis model based on the Amadal's law. To the best of our knowledge, we are the first to formalize a practical model for measuring concurrent structures' speedup. To demonstrate our idea, we also build some edge-cutting BSTs following our principles, testing them under different workloads. The result provides compelling evidence to back the our guidelines, and shows that our theory is useful for reasoning the varied performance.

1 Introduction

As multi-core chips are widely used in commodity devices, designing concurrent structures has become a hot topic. These years, researchers focus on designing concurrent BSTs. Normally, sequential BSTs should be entirely locked when accessed by multi-threads. Concurrent BSTs leverage the property that modifications naturally happen in disparate places, therefore using finer-grained locks or flags could boost the parallelism. To improve the performance of concurrent BSTs, there are several aspects of optimization. From the perspective of hardware interface, we could apply varied atomic operations, like compare-and-swap[14], fetch-and-add[4]. By the help of underlying system support, we can devise RCU[9] and STM[12]. From the perspective of structures, external trees and internal trees are both available. To achieve greater disjoint-parallelism[7], the locks which previous on the nodes could be moved to the edges.

ASCYLIB[2] is a concurrent structure library, including a bunch of different structures such as linked-list, hash-table, and BSTs. The core of ASCY is that concurrent structures should resemble their sequential counterparts. The author addresses that structures follow ASCY-compliant pattern use less power consumption, and achieve portable scalability that scale well under different workloads and platforms.

In this paper, we adopt the similar idea as ASCY to implement different concurrent BSTs. We compare their performance under various workloads and platforms, and propose our own principles of designing concurrent structures. Based on the Amdahl's law[6], we present the first model to analyze speedup for concurrent structures.

2 Preliminary

We view the BST as a dictionary to retrieve unique key-value pairs. There are three kinds of operations with the dictionary, we define them as follow:

- Search(key). It calls $find$ operation to reach the corresponding leaf node, and returns $true$ if the key matches, or it returns $false$.

- Insert(key). It begins to reach the candidate leaf node, if there's already such a key, it returns $false$. Otherwise it adds the key into the dictionary.

- Delete(key). It begins to reach the candidate leaf node, if there's no such a key, it returns $false$. Otherwise it removes the key from the dictionary.

To formalize the operations from the perspective of thread interactions, we propose a interface of our design in figure 1. Each operation will take "snapshot" of the tree by the $find$ routine, and adopt different consistency controller to manage contentions and start retry. The similar idea is used by [5], to facilitate the performance under a little modification, the author use simple locks with some checks to develop a concurrent skiplist.

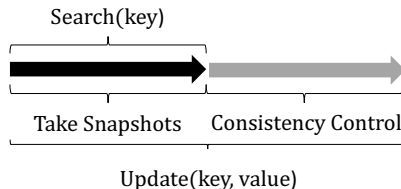There are generally two types of structures in BSTs, one is the internal tree, the same as sequential BSTs;

Figure 1: Operation Interface

the other is the external tree, using more space but reduces contention to the leaf nodes. Figure 2 shows the

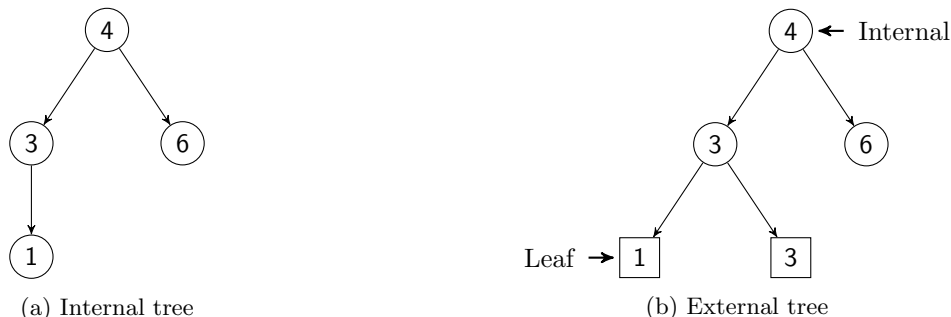(a) Internal tree

(b) External tree

Figure 2: Two types of BSTs

different structures, where the external tree stores keys only in leaves, but the internal tree stores keys in both. The internal tree will be scaled up as the key buckets growing, whereas external tree renders better performance when the key range is small[11]. We use the external tree to clearly our idea, and we believe that the same technique could be applying to the internal tree with little adapt. Our FEM-BST, using flag and mark indicators, is also very friendly to be adjusted into a lock-free version.

To avoid some special situations, we confine the key range in $(-\infty, \infty)$ as describe in [3]. Figure 3 shows the initial structure, it is guaranteed that the initial three nodes will never be removed. As illustrate in table
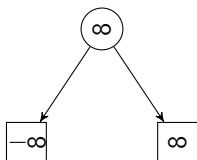

Figure 3: Initial structure of our BST

1, we implement 5 kinds of BSTs using different locks, all of which are CAS-based locks. Furthermore, we implement different consistency controllers to check whether the state is consistent during the modification. The ticket lock uses two numbers, ticket and version. The $version$ is to record the current version of the node, the $ticket$ is used for lock. If the current $ticket$ is not equal to the $version$, it indicates that the node is locked. The flag-marked lock simply use two boolean field. The flag field is to indicate whether the node is owned by a thread or not, the marked field is to denote whether the node is under the delete operation. In this paper, we introduce the algorithm of FEM-BST. It has locks on nodes, but the performance is no different as the formal edge-based locks. We design fine checking mechanism to ensure the correctness and improve parallelism.

2

Name	Lock
BST	none
SYN-BST	synchronized
FN-BST	flag-based lock on node
FE-BST	flag-based lock on edge
FEM-BST	flag-mark-based lock on edge
TN-BST	ticket lock on node

Table 1: The variant BSTs

3 Algorithm

3.1 Search

We use *ppred* to denote the grandparent node, *pred* to denote the parent node, and *curr* for the current node. Moreover, *pright* and *right* represent the directions. As shown in the algorithm $FIND$, the operation goes from the root node to the corresponding leaf node, and returns a snapshot which includes 5 elements: $\{ppred, pright, pred, right, curr\}$ in figure 4. The search algorithm is based on the optimistic strategy, which

---

Algorithm 1 Find

1: $curr \leftarrow root$
2: $ppred, pred \leftarrow null$
3: **while** $curr \neq leaf$ **do**
4:     taking snapshot
5:     **if** $curr.key < key$ **then**
6:         $curr \leftarrow curr.left$
7:     **else**
8:         $curr \leftarrow curr.right$
9:     **end if**
10: **end while**
11: **Return** *snapshot*

---

finds the node *was* in the tree on the search path, other than the node *is* currently in the tree. In fact, we can hardly implement such an algorithm that returns the result at the exact time-stamp.
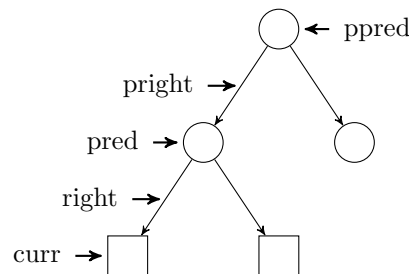

Figure 4: Take snapshot from top to down

3.2 Insert

To begin with, the insert operation gets the snapshot in line 2. It compares whether the key is equal to the *curr* node's. If there's such a key in the tree, it returns $false$. Otherwise it tries to lock the node, and handles some inconsistency situations. Figure 5 shows two inconsistency situations, one is that the *pred* node is $marked(line\ 9)$, which indicates there's another operation deleting the *pred* node; the other is when

the *pred* node is not linked to the *curr* node(*line* 13). Either of the situation indicates the operation has to retry to find the new corresponding node. There's no need to retry if the parent node is only locked, since it indicates that there's another node inserted in the other side while does not affect the current operation. The insert operation is guaranteed to be succeed when it locks the current node. Finally it constructs the node, and releases the lock.

Algorithm 2 Insert

1: **while** $TRUE$ **do**
2:     $\{curr, pred, right\} \leftarrow find(key)$
3:     **if** $curr.key == key$ **then**
4:         **Return** $FALSE$
5:     **end if**
6:     **if** $!curr.tryLock()$ **then**
7:         **Continue**
8:     **end if**
9:     **if** $pred.marked$ **then**   ▷ parent node already deleted
10:         $curr.release()$
11:         **Continue**
12:     **end if**
13:     **if** $right$ AND $pred.right \neq curr$ OR $!right$ AND $pred.left \neq curr$ **then**
14:         $curr.release()$
15:         **Continue**
16:     **end if**
17:     Construct $newParent$ and $insertNode$
18:     **if** $right$ **then**   ▷ according to right flag
19:         $pred.right \leftarrow newParent$
20:     **else**
21:         $pred.left \leftarrow newParent$
22:     **end if**
23:     $curr.release()$
24:     **Return** $TRUE$
25: **end while**

3.3   Delete

Like the insert operation, the delete operation starts by getting the snapshot in Algorithm Delete *line* 2. However, it has to get an extra *ppred* node and *pRight* indicator, as it should move the grandparent's link to the sibling of the removed node. It first compares the key to the current node, if it is not equal, it returns $false$(*line* 4). Otherwise it first tries to lock the *pred* node(*line* 7). It then checks the *ppred* node's state(*line* 11), ensuring the grandparent is neither marked or linked to another node. If it successes, it then begins to lock the current node(*line*16). Both the above locks has to be released once it detects inconsistency. After locking the parent node and the current node, it has to wait for the operation upon *siblingnode* to be finished(*line* [34 − 48]). The delete operation will successfully remove the node from the tree when it gets the correct sibling.

Figure 6 shows the two situations the algorithm *Delete* has to retry. Both happen on the grandparent node. For the situation b, we have to first check whether the node is released or not. Because once the sibling node is released, it could not be locked again when the parent node is locked. Therefore we do not have to retry from the root node.
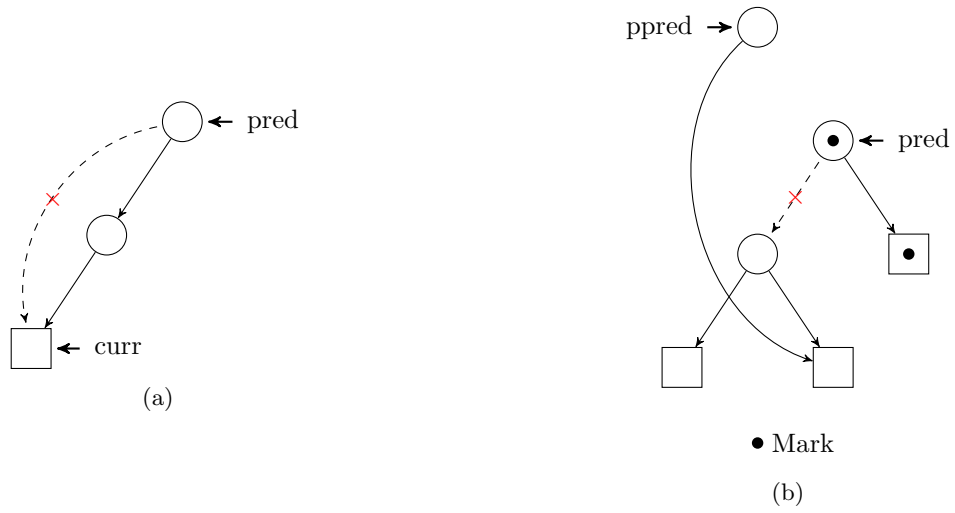
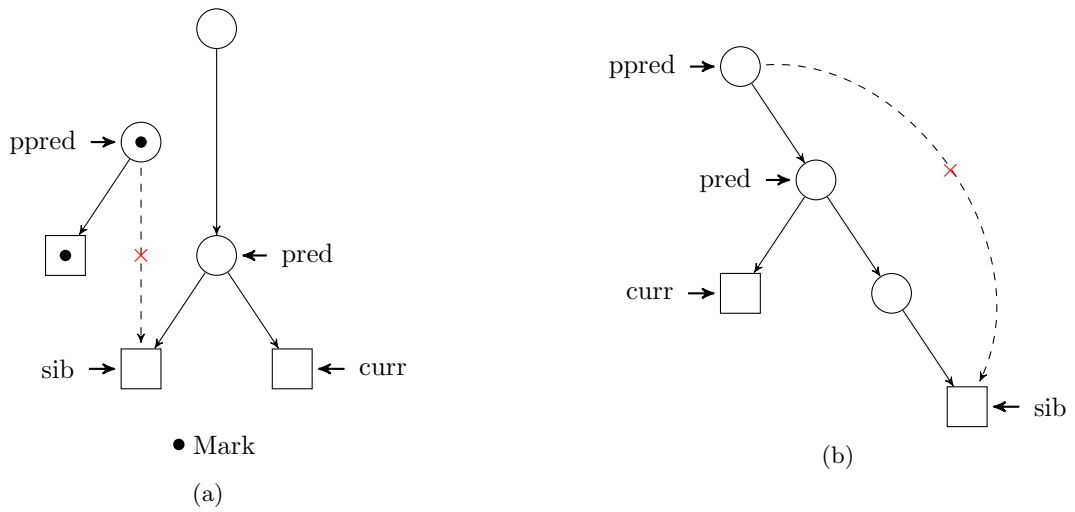Figure 5: Two wrong situations of insert operation

Figure 6: Two wrong situations of delete operation

Algorithm 3 Delete

1: **while** $TRUE$ **do**
2:      $\{curr, pred, ppred, right, pright\} \leftarrow find(key)$
3:      **if** $curr.key \neq key$ **then**
4:         **Return** $FALSE$
5:      **end if**
6:      Construct new $node$
7:      **if** $pred.marked$ OR $!pred.tryLock()$ **then**   ▷ check parent node
8:         **Continue**
9:      **else**
10:         $pred.marked = TRUE$
11:         **if** $ppred.marked$ OR $pRight$ AND $ppred.right \nleftarrow pred$ OR $!pRight$ AND $ppred.left \neq pred$ **then**
     ▷ check grandparent node
12:             $pred.marked \leftarrow FALSE$
13:             $pred.relase()$
14:             **Continue**
15:         **end if**
16:         **if** $!curr.tryLock()$ **then**   ▷ check curr node
17:             $pred.marked \leftarrow FALSE$
18:             $pred.release()$
19:             **Continue**
20:         **else**
21:             $curr.marked \leftarrow TRUE$
22:             **if** $right$ AND $pred.right \neq curr$ OR $!right$ AND $pred.left \neq curr$ **then**
23:                 $curr.marked \leftarrow false$
24:                 $curr.release()$
25:                 $pred.marked \leftarrow false$
26:                 $pred.release()$
27:                 **Continue**
28:             **end if**
29:             **if** $right$ **then**   ▷ get sibling node
30:                 $node \leftarrow pred.left$
31:             **else**
32:                 $node \leftarrow pred.right$
33:             **end if**
34:             **while** $TRUE$ **do**
35:                 **if** $right$ **then**
36:                     **if** $node.lock$ OR $pred.left \neq node$ **then**   ▷ The order cannot be changed
37:                         $node \leftarrow pred.left$
38:                         **Continue**
39:                     **end if**
40:                     **Break**
41:                 **else**
42:                     **if** $node.lock$ OR $pred.left \neq node$ **then**   ▷ The order cannot be changed
43:                         $node \leftarrow pred.left$
44:                         **Continue**
45:                     **end if**
46:                     **Break**
47:                 **end if**
48:             **end while**
49:         **end if**
50:      **end if**

```
51:        if pRight then
52:            ppred.right ← node
53:        else
54:            ppred.left ← node
55:        end if
56:        Return TRUE
57: end while
```

4 Correctness

We first prove that our FEM-BST maintains the property during executions, then prove it is deadlock free, and point out the linearization points. The proof structure is similar as [10].

4.1 maintain structure

We prove that the following invariants hold during the modification:

1. The root node is never removed.

2. The key field never changes.

3. The left child's key is always less than the parent node, the right child's key is always greater or equal to the parent node.

4. Once a parent node is locked and checked, the insert operation must succeed.

5. Once the parent node and current node are locked and checked, the delete operation must succeed.

Proof:

1. The available key range is in $(-\infty, \infty)$, therefore the initial three nodes will never be retrieved by modifications.

2. An insert operation is finished by constructing two new nodes, linked with the existing node; A delete operation is finished by moving the grandparent pointer to the sibling of removed node. Hence the key field never changes.

3. Any modification upon the BST take a correct snapshot at a specific time-stamp, hence the *curr* node must be a child of the *pred* node. For a insert operation, the newly construct node follows the right direction; for a delete operation, the grandparent's pointer pointed to the existed child in the tree.

4. The insert operation first tries to lock the *curr* node. After locking, no other operation could take upon the *curr* node. It then checks whether the *pred* node is marked. If it is marked, the *curr* lock is released, the operation retries to find a new corresponding node. Otherwise the insert operation will succeed.

5. The delete operation tries to lock the node in the following order: $pred- > curr$. locks on the *pred* node and *curr* node ensure that other thread could see the marked status, thereby do not affect the current deletion. The *sibling* node will finally be in a clean state, since there's finite set of modify operations.

4.2 liveness

We prove our FEM-BST is deadlock-free.
An important observation is that insert and delete operations locks the node from top to down order. The insert operation only locks the *curr* node. The delete operation first locks the *pred* node, and release it once it detects the child is locked. Therefore once contention is detected, the operations will be rolled back.
Another contention happens in the delete operation is when it tries to lock the sibling node. An important

observation is that insert and delete operations locks the node from top to down order. Because there's finite insert and delete operation, and any operation locks the node from top to down order, the *pred* and *cur* will not be violated. Finally the delete operatoin could get the sibling in isolation. Thus the FEM-BST is deadlock-free.

4.3   linearization point

- Insert. The linearization point of a successful insert operation is in Algorithm 2 *line* 6; the linearization point of a failure insertion operation is in Algorithm 2 *line* 4.

- Delete. The linearization point of a successful delete operation is in Algorithm 3 *line* 6 and *line* 10; the linearization point of a failure delete operation is in Algorithm 3 *line* 4.

5   Performance

All of our BSTs in table 1 are implemented by Java, JDK 1.7. We set up the test by randomly inserting $\frac{bucket}{2}$ elements into the tree, and then running cases threads for 5s to ensure that elements are inserted and deleted multiple times. A similar idea is used in [1]. Experiments are performed on the platform of two Intel E5-2680 processors, 32 hardware threads with hyper-thread supported. The system is Red Hat Enterprise Linux Server release 6.3.

We compared different BSTs with respect to throughput, which is defined as the total number of operations completed per second. The number of threads was set from 1 to 32 and the bucket size is 10000 and 100000. We use two workloads: low-contention: 9% insert, 1% delete, 90%search, and mid-contention: 20% insert, 10% delete, 70% search. Figure 7 shows the result under different modification distribution. The result
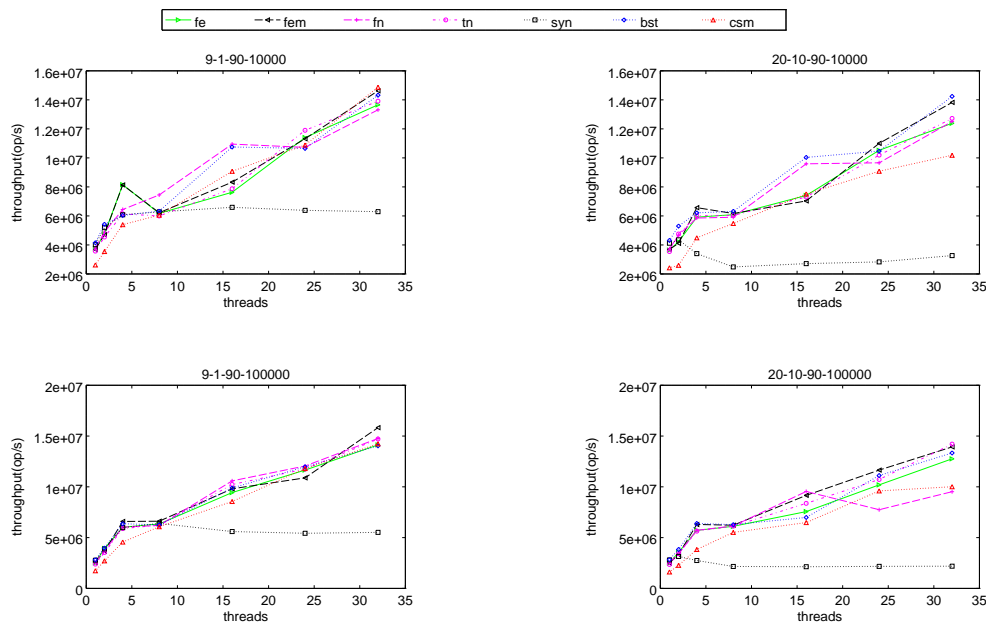

Figure 7: Comparison of throughput of different BSTs under varied workload

shows that our BSTs render similar performance as the unsynchronized-BST, which is used to stand for a possible upper-bound. The FN-BST is the least scalable one, since it need to lock more nodes than edge based BSTs. It is also worse than the TN-BST, which has to get little snapshot during searching. Our FEM-BST has the best performance, since it handles every possible inconsistency. We conclude our theory in the following part.

8

6   Principles

We propose some concurrent data structure design principles according to our experiment and the interface in figure 1:

1. To achieve correctness, we must ensure either the snapshot or consistency controller exists.

2. For the general lock-based algorithms, the more the amount of snapshot, the less complexity of consistency controller.

3. For the lock-free algorithms, we might obtain both complex snapshot and consistency controller.

4. Snapshot and consistency work together to affect single-thread performance. The higher the single-thread performance, the lower the progress conditions (parallelism).

Table 2 lists out our analyze of common concurrent techniques. *Sycrhonized* is for a coarse-grained version of sequential data structure, which handles the contention by locking the whole object. *STM* utilizes parallelism by optimistic control strategy, and it should obtain a large amount of snapshot. *TicketLock* is mentioned above, which implements the lock by version numbers. *FairLock* represents fine-grained locks use a queued structure. *NonFairLock* is implemented by flags. *Lockfree* algorithms usually need very detailed design with thread interactions. *Waitfree* algorithms are more strict than *Lockfree* in progress condition, the only known waitfree structures are queue[8] and linked-list[13].

Techniques	Parallelism	Snapshot	Consistency Controller
Sycrhonized	very low	very low	high
STM	low	medium	medium
TicketLock	medium	high	low
FairLock	low	low	high(AQS framework)
NonFairLock	medium	low	high
Lockfree	medium	medium	high
Waitfree	high	medium	very high

Table 2: Concurrent Techniques Comparison

7   Model

To the best of our knowledge, there's no any practical model fit for measuring the speedup of concurrent data structures. Here we present an analysis model to transform the initial amadal law for concurrent structures.

$$speedup = \frac{1}{(1-p) - p/n}$$

The above equation is the most common known form of Amadal's law, where p is the parallel ratio of a program. We assume $p = 1$ in concurrent structures, thereby the traditional model needs to be modified. We start from comparing the *workload* of sequential part(w_s) and parallel part(w_p) of sequential structure to the concurrent structure where the parallel workload(w_p') is different.

$$speedup = \frac{w_s + w_p}{w_s + w_p'/P}$$
$$= \frac{w_p}{w_p'/P}$$

In the original equation, P is defined as the number of processors, however in the concurrent structure, it is nearly impossible that all of the threads are taking into effect. Therefore we define $P(w_{snapshot}, w_{control})$ as
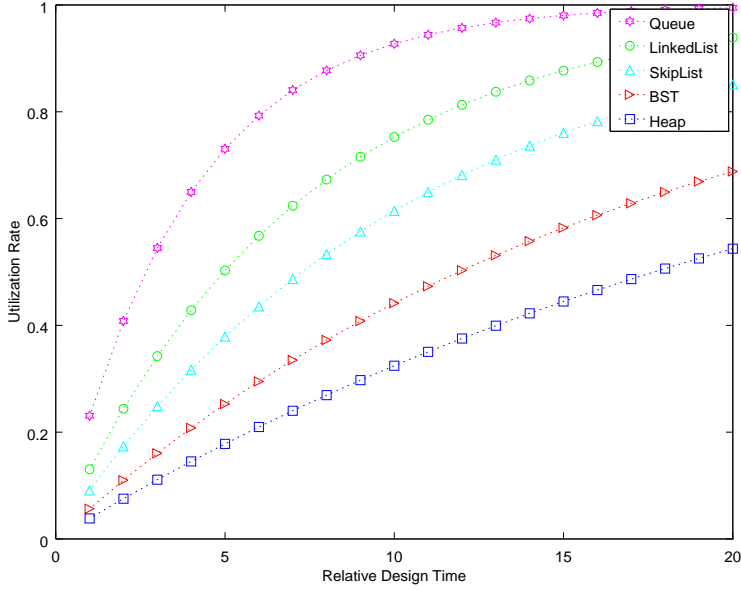
Figure 8: $\alpha(t) = \frac{w_{snapshot} * \beta}{w_{control}}(-h^{-t} + 1), 1 < h$(hardness of the structures)

a function which represents real parallelism. Furthermore, since new operations, snapshot and consistency control, are involved in the parallel version, we define $W_p(w_{snapshot}, w_{control})$ as the new parallel workload.

$$W_p(w_{snapshot}, w_{control}) = w_p + w_{snapshot} + w_{control}$$
$$P(w_{snapshot}, w_{control}) = P * (1 - c) * \alpha$$
$$0 \le c \le 1$$

Where c stands for the contention rate, α is the rate of taking effects on linearization points, β is the rate of recording valid linearization points, they are algorithm related variables.

$$speedup = \frac{P * (1 - c) * \alpha}{1 + \frac{w_{snapshot}}{w_p} + \frac{w_{control}}{w_p}}$$
$$\frac{1}{w_{snapshot}} \le \beta \le 1$$
$$0 \le \alpha \le \frac{w_{snapshot} * \beta}{w_{control}} \le 1$$

The α factor is associated with the hardness of the sequential structure, where we define the hardness is proportional to the amount of adjust of the sequential part. Hence, the greater the a element has to communicate with others, the harder the structure, which means it needs more time to raise α. Figure 8 demonstrates our measure of α factor. We have to pay much more amount of effort into "hard" structures. For instance, for the heap, we have to lock the whole path from root to leaf during modifications. Hence, to relax such a adjustment is difficult. However queue only need to modify the tail and head, therefore is easier to raise parallelism.

8 Conclusion

We present a pattern of design concurrent data structures with a model to formalize the speedup measure. We also provide compelling evidence by measuring different kinds of BSTs under various workloads. An

immediate discussion in the future would be implementing other structures such as skip-lists and heaps to illustrate our model. Another topic is to refine our model to measure the speedup accurately, and develop a software for practice.

References

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. Technical report, Technical Report MSR-TR-2014-16, Microsoft Research, 2014.

[2] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644. ACM, 2015.

[3] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140. ACM, 2010.

[4] Allan Gottlieb and Clyde P Kruskal. Coordinating parallel processors: A partial unification. *ACM SIGARCH Computer Architecture News*, 9(6):16–24, 1981.

[5] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity*, pages 124–138. Springer, 2007.

[6] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, (7):33–38, 2008.

[7] CAR Hoare. *Parallel programming: an axiomatic approach.* Springer, 1976.

[8] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *ACM SIGPLAN Notices*, volume 46, pages 223–234. ACM, 2011.

[9] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[10] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM, 1996.

[11] Arunmoezhi Ramachandran and Neeraj Mittal. Castle: fast concurrent internal binary search tree using edge-based locking. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 281–282. ACM, 2015.

[12] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[13] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Principles of Distributed Systems*, pages 330–344. Springer, 2012.

[14] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.