Range Queries in Non-blocking k-ary Search Trees

Trevor Brown[1], Hillel Avni[2]

[1]Dept. of Computer Science, University of Toronto
tabrown@cs.toronto.edu
[2]Dept. of Computer Science, Tel-Aviv University
hillel.avni@gmail.com

Abstract. We present a linearizable, non-blocking k-ary search tree (k-ST) that supports fast searches and range queries. Our algorithm uses single-word compare-and-swap (CAS) operations, and tolerates any number of crash failures. Performance experiments show that, for workloads containing small range queries, our k-ST significantly outperforms other algorithms which support these operations, and rivals the performance of a leading concurrent skip-list, which provides range queries that cannot always be linearized.

1 Introduction and Related Work

The ordered set abstract data type (ADT) represents a set of keys drawn from an ordered universe, and supports three operations: INSERT(key), DELETE(key), and FIND(key). We add to these an operation RANGEQUERY(a, b), where $a \leq b$, which returns all keys in the closed interval $[a, b]$. This is useful for various database applications.

Perhaps the most straightforward way to implement this ADT is to employ software transactional memory (STM) [13]. STM allows a programmer to specify that certain blocks of code should be executed atomically, relative to one another. Recently, several fast binary search tree algorithms using STM have been introduced [7, 2]. Although they offer good performance for INSERTs, DELETEs and FINDs, they achieve this performance, in part, by carefully limiting the amount of data protected by their transactions. However, since computing a range query means protecting all keys in the range from change during a transaction, STM techniques presently involve too much overhead to be applied to this problem.

Another simple approach is to lock the entire data structure, and compute a range query while it is locked. One can refine this technique by using a more fine-grained locking scheme, so that only part of the data structure needs to be locked to perform an update or compute a range query. For instance, in leaf-oriented trees, where all keys in the set are stored in the leaves of the tree, updates to the tree can be performed by local modifications close to the leaves. Therefore, it is often sufficient to lock only the last couple of nodes on the path to a leaf, rather than the entire path from the root. However, as was the case for STM,

a range query can only be computed if every key in the range is protected, so typically every node containing a key in the range must be locked.

Persistent data structures [11] offer another approach. The nodes in a persistent data structure are immutable, so updates create new nodes, rather than modifying existing ones. In the case of a persistent tree, a change to one node involves recreating the entire path from the root to that node. After the change, the data structure has a new root, and the old version of the data structure remains accessible (via the old root). Hence, it is trivial to implement range queries in a persistent tree. However, significant downsides include contention at the root, and the duplication of many nodes during updates.

Brown and Helga [9] presented a k-ST in which each internal node has k children, and each leaf contains up to $k-1$ keys. For large values of k, this translates into an algorithm which minimizes cache misses and benefits from processor pre-fetching mechanisms. In some ways, the k-ST is similar to a persistent data structure. The keys of a node are immutable, but the child pointers of a node can be changed. The structure is also leaf-oriented, meaning that all keys in the set are stored in the leaves of the tree. Hence, when an update adds or removes a key from the set, the leaf into which the key should be inserted, or from which the key should be deleted, is simply replaced by a new leaf. Since the old leaf's keys remains unmodified, range queries using this leaf need only check that it has not been replaced by another leaf to determine that its keys are all in the data structure. To make this more efficient, we modify this structure by adding a *dirty*-bit to each leaf, which is set just before the leaf is replaced.

Braginsky and Petrank [5] presented a non-blocking B$^+$tree, another search tree of large arity. However, whereas the k-ST's nodes have immutable keys, the nodes of Braginsky's B$^+$tree do not. Hence, our technique for performing range queries cannot be efficiently applied to their data structure.

Snapshots offer another approach for implementing range queries. If we could quickly take a snapshot of the data structure, then we could simply perform a sequential range query on the result. The snapshot object is a vector V of data elements supporting two operations: UPDATE(i, val), which atomically sets V_i to val, and SCAN, which atomically reads and returns all of the elements of V. SCAN can be implemented by repeatedly performing a pair of COLLECTs (which read each element of V in sequence and return a new vector containing the values it read) until the results of the two COLLECTs are equal [1]. Attiya, et al. [3] introduced *partial snapshots*, offering a modified SCAN$(i_1, i_2, ..., i_n)$ operation which operates on a subset of the elements of V. Their construction requires both CAS and fetch-and-add.

Recently, two high-performance tree structures offering $O(1)$ time snapshots have been published. Both structures use a lazy copy-on-write scheme that we now describe.

Ctrie is a non-blocking concurrent hash trie due to Prokopec et al. [12]. Keys are hashed, and the bits of these hashes are used to navigate the trie. To facilitate the computation of fast snapshots, a sequence number is associated with each node in the data structure. Each time a snapshot is taken, the root is copied and

its sequence number is incremented. An update or search in the trie reads this sequence number *seq* when it starts and, while traversing the trie, it duplicates each node whose sequence number is less than *seq*. The update then performs a variant of a double-compare-single-swap operation to atomically change a pointer while ensuring the root's current sequence number matches *seq*. Because keys are ordered by their hashes in the trie, it is hard to use Ctrie to efficiently implement range queries. To do so, one must iterate over all keys in the snapshot.

The second structure, Snap, is a lock-based AVL tree due to Bronson et al. [6]. Whereas Ctrie added sequence numbers, Snap *marks* each node to indicate that it should no longer by modified. Updates are organized into *epochs*, with each epoch represented by an object in memory containing a count of the number of active updates belonging to that epoch. A snapshot marks the root node, ends the current epoch, and blocks further updates from starting until all updates in the current epoch finish. Once updates are no longer blocked, they copy and mark each node they see whose parent is marked. Like Ctrie, this pushes work from snapshots onto subsequent updates. If these snapshots are used to compute small range queries, this may result in excessive duplication of unrelated parts of the structure.

If we view shared memory as a contiguous array, then our range queries are similar to *partial snapshots*. We implement two optimizations specific to our data structure. First, when we traverse the tree to perform our initial COLLECT, we need only read a pointer to each *leaf* that contains a key in the desired range (rather than reading each key). This is a significant optimization when k is large, e.g., 64. Second, instead of performing a second COLLECT (which would involve saving the parent of each leaf or traversing the tree again), we can simply check the *dirty*-bit of each node read by the first COLLECT. As a further optimization, range queries can return a sequence of leaves, rather than copying their keys into an auxiliary structure.

Contributions of this work:

- We present a new, provably correct data structure, and demonstrate experimentally that, for two very different sizes of range queries, it significantly outperforms data structures offering $O(1)$ time snapshots. In many cases, it even outperforms a non-blocking skip-list, whose range queries cannot always be linearized.
- We contribute to a better understanding of the performance limitations of the $O(1)$ time snapshot technique for this application.

The structure of the remainder of this paper is as follows. In Sec. 2, we describe the data structure, how updates are performed, and our technique for computing partial snapshots of the nodes of the tree. We give the details of how range queries are computed from these partial snapshots of nodes in Sec. 3. A sketch of a correctness proof is presented in Sec. 4. (The full version of this paper [8] contains a detailed proof.) Experimental results are presented in Sec. 5. Future work and conclusions are discussed in Sec. 6.

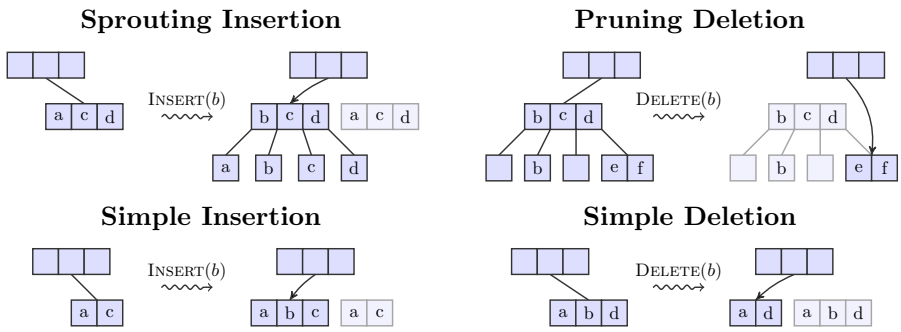

Fig. 1. the four k-ST update operations.

2 Basic Operations and Validate

The k-ST is a linearizable, leaf-oriented search tree in which each internal node has k children and each leaf contains up to $k-1$ keys. Its non-blocking operations, Find, Insert and Delete, implement the set ADT. Find(key) returns True if key is in the set, and False otherwise. If key is not already in the set, then Insert(key) adds key and returns True. Otherwise it returns False. If key is in the set, then Delete(key) removes key and returns True. Otherwise it returns False. The k-ST can be extended to implement the dictionary ADT, in which a value is associated with each key (described in the technical report for [9]). Although a leaf-oriented tree occupies more space than a node-oriented tree (since it contains up to twice as many nodes), the expected depth of a key or value is only marginally higher, since more than half of the nodes are leaves. Additionally, the fact that all updates in a leaf-oriented tree can be performed by a local change near the leaves dramatically simplifies the task of proving that updates do not interfere with one another.

Basic *k*-ST operations These operations are implemented as in [9]. Find is conceptually simple, and extremely fast. It traverses the tree just as it would in the sequential case. However, concurrent updates can prevent its termination (see [10]). Insert and Delete are each split into two cases, for a total of four update operations: sprouting insertion, simple insertion, pruning deletion and simple deletion (see Fig. 1). An Insert into leaf l will be a sprouting insertion when l already has $k-1$ keys. Since l cannot accommodate any more keys, it is atomically replaced (using CAS) by a newly created sub-tree that contains the original $k-1$ keys, as well as the key being inserted. Otherwise, the Insert will be a simple insertion, which will atomically replace l with a newly created leaf, containing the keys of l, as well as the key being inserted. A Delete from leaf l will be a pruning deletion if l has one key and exactly one non-empty sibling. Otherwise, the Delete will be a simple deletion, which will atomically replace l with a newly created leaf, containing the keys of l, except for the key being

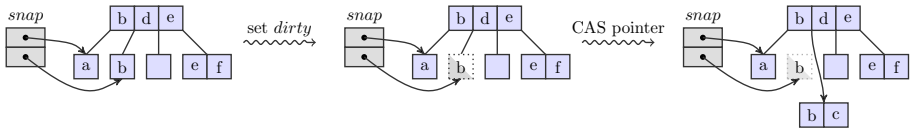

Fig. 2. A VALIDATE is in progress on the leaves containing a and b. Before it can check the *dirty* bits of these leaves at line 2, an update INSERT(c) sets the *dirty* bit of the leaf containing b. After this, the VALIDATE is doomed to return FALSE. The INSERT(c) then changes its child pointer and finishes.

deleted. Note that if l has one key, then l will be empty after a simple deletion. With this set of updates, it is easy to see that the keys of a node never change, and that internal nodes always have $k-1$ keys and k children.

The non-blocking progress property is guaranteed by a helping scheme that is generalized from the work of Ellen et al. [10], and is somewhat similar to the cooperative technique of Barnes [4]. Every time a process p performs an update U, it stores information in the nodes it will modify to allow any other process to perform U on p's behalf. When a process is prevented from making progress by another operation, it helps that operation complete, and then retries its own.

Validate We include a VALIDATE subroutine, which is used in the RANGEQUERY algorithm in Sec. 3. It is closely related to a partial snapshot operation when memory is viewed as an array of locations. VALIDATE takes as argument a sequence of pointers to leaves which were reached by following child pointers from the root.

```
1  VALIDATE(p_1, p_2, ..., p_n) {
2    check the dirty bit of each leaf pointed to by an element of {p_1, ..., p_n}
3    if any dirty bit is TRUE {
4      return FALSE
5    } else {
6      return TRUE
7  } }
```

The *dirty* field is a single bit included in each leaf l that is initially FALSE, and is irrevocably set to TRUE just before any CAS that will remove or replace l. (See Fig. 2 for an illustration of how a *dirty* bit causes a snapshot to retry.) Thus, a FALSE *dirty* bit in a leaf l that was visited in the tree implies that l is in the tree. Hence, if each $l \in \{p_1, ..., p_n\}$ satisfies $l.dirty =$ FALSE, then we know each l was in the tree when the its *dirty* bit was read at line 2 by the algorithm. Furthermore, since the keys of a node never change, l's FALSE *dirty* bit means that all of the keys in l are in the tree. Hence, if VALIDATE returns TRUE, all of the keys contained in the leaves in the snapshot are in the tree when the VALIDATE began. If VALIDATE returns FALSE, then there is a leaf that has been or is in the process of being replaced.

```
8   type Node {
9     Key ∪ {∞}  a_1, ..., a_{k-1}
10  }

11  subtype Internal of Node {
12    Node  c_1, ..., c_k
13  }

14  subtype Leaf of Node {
15    boolean dirty
        ▷ (initially FALSE)
16  }

17  RANGEQUERY(Key lo, Key hi) : List of Nodes {
      ▷ Precondition: lo, hi ≠ ∞, and lo ≤ hi
18    List snap := new List()
      ▷ DFS to populate snap with all leaves that could possibly contain a key in [lo, hi]
19    Stack s := new Stack()
20    s.push(root.c_1)
21    while |s| > 0 {
22      Node u := s.pop()
23      if u is a Leaf then do snap.add(u)
        ▷ Determine which children of u to traverse
24      int i := 1
25      while i < k and u.a_i ≤ lo {          ▷ Find first sub-tree that intersects [lo, ∞)
26        i := i + 1
27      }
28      if i = 1 {                            ▷ Boundary case: don't test u.a_0 below
29        s.push(u.c_i)
30        i := i + 1
31      }
32      while i ≤ k and u.a_{i-1} ≤ hi {      ▷ Push sub-trees until all keys are in (hi, ∞)
33        s.push(u.c_i)
34        i := i + 1
35    } }
      ▷ Validate (check the nodes in snap have not changed)
36    if not VALIDATE(snap) then retry (i.e., go back to line 18)
      ▷ Return all leaves in snap that contain some key in range [lo, hi]
37    List result := new List()
38    for each u in snap {
39      if at least one of u's keys is in range [lo, hi] then do result.add(u)
40    }
41    return result
42  }
```

Fig. 3. Abridged type definitions and pseudocode for RANGEQUERY. RANGEQUERY accepts two keys, lo and hi, as arguments, and returns all leaves that (a) were in the tree at the linearization point, and (b) have a key in the closed interval $[lo, hi]$.

3 Range Queries in a k-ST

An abridged description of the type definitions of the data structure and Java-like pseudocode for the RANGEQUERY operation are given in Fig. 3. We borrow the concept of a *reference* type from Java. In this psuedocode, variables of any type $E \notin \{\text{int, boolean}\}$ are references to objects of type E. A reference x is like pointer, but is automatically dereferenced when a field of the object is accessed with the (.) operator, as in: $x.field$ (which means the same as `x->field` in C). References take on the special value NULL when they do not point to any object. (However, no field of any node is ever NULL.)

We now take a high-level tour through the RANGEQUERY algorithm. The algorithm begins by declaring a list *snap* at line 18 to hold pointers to all leaves which may contain a key in $[lo, hi]$. In lines 19-35 the algorithm traverses the

tree, saving pointers in *snap*. It uses a depth-first-search (DFS), implemented with a stack (instead of recursion), except that it may prune some of the children of each node, and avoid pushing them onto the stack. The loop at line 25 prunes those children that are the roots of sub-trees with keys strictly less than lo. The loop at line 32 then pushes children onto the stack until it hits the first child that is the root of a sub-tree with keys strictly greater than hi. Both of these loops use the fact that keys are maintained in increasing order within each node. It follows that all paths that could lead to keys in $[lo, hi]$ are explored, and all terminal leaves on these paths are placed in *snap* at line 23.

RangeQuery then calls Validate (described in the previous section). If this validation is successful, then each element of *snap* that points to a leaf containing at least one key in $[lo, hi]$ is copied into *result* by the loop at lines 38-40. The range query can be modified to return a list of keys instead of nodes simply by changing the final loop (since the keys of a node never change).

4 Correctness

We now provide a proof sketch. The interested reader can find the details of the following results in the full version of this paper [8].

Observation 1. *Apart from child pointers, nodes are never modified. To insert or delete a key,* Insert *and* Delete ***replace*** *affected node(s) with newly created node(s).*

Lemma 2. *If no* Insert *or* Delete *operations are executing, then there are no dirty leaves reachable by following child pointers from root.*

The proof of this lemma is quite laborious, but the intuition is simple. Leaves only become dirty (have their dirty bit set) by an update (Insert or Delete) right before the update executes a *Child CAS*, which changes a child pointer to remove the leaf from the tree.

Next, we prove progress. Since RangeQuery does not write to shared memory, it cannot affect the correctness or progress of Find, Insert or Delete. The proof in [9] still applies, and Find, Insert and Delete are all non-blocking. Hence, it is sufficient to prove that, if a RangeQuery operation is performed, it eventually terminates if no Insert or Delete operations are executing. If a RangeQuery is being performed and no Insert or Delete operations are executing then, by Lemma 2, RangeQuery will eventually encounter no dirty leaf and, hence, will eventually terminate.

Theorem 3. *All operations are non-blocking.*

Definition 4. *At any configuration C, let T_C be the k-ary tree formed by the child references. We define the* ***search path*** *for key a in configuration C to be the unique path in T_C that would be followed by the ordinary sequential k-ST search procedure.*

Definition 5. *We define the* ***range*** *of a leaf u in configuration C to be the set R of keys such that, for any key $a \in R$, u is the terminal node on the search path for a in configuration C. (Consequently, if u is not in the tree, its range is the empty set.)*

To simplify the statements of upcoming results, we also make two more simple definitions. A node is ***in the tree*** if it is reachable by following child pointers from the root. A node is ***initiated*** if it has ever been in the tree.

Lemma 6. *If an initiated leaf is not in the tree, then it is dirty.*

This simple result follows from the fact that any leaf removed from the tree is removed by a *Child CAS*, just prior to which the leaf's dirty bit is set.

Lemma 7. *If a key is inserted into or deleted from the range of a leaf u, and u is in the tree just before the linearization point of the* Insert *or* Delete*, then u is no longer in the tree just after the linearization point.*

This lemma relies heavily on results from [9]. We first prove that a successful update on key a must remove a leaf whose range contained a at the linearization point of a Find which the update invokes. We then prove that this leaf's range must still contain a just before the update is linearized. Finally, we invoke the k-ary search tree property to argue that a can only be in the range of one leaf just before the update is linearized, which implies the result.

Now we can prove the correctness of RangeQuery. We linearize each completed invocation of RangeQuery immediately before performing Validate for the last time.

Theorem 8. *Each invocation of* RangeQuery(lo, hi) *returns a list containing precisely the set of leaves in the tree that have a key in the range $[lo, hi]$ at the time the* RangeQuery *is linearized.*

This final result is proved with the help of Lemma 6, Lemma 7, and the following sub-claims:

(a) In every configuration C, every internal node has exactly k children and satisfies the k-ary search tree property.
(b) The DFS in the first loop of RangeQuery traverses the relevant part of the tree and adds every leaf it visits to *snap*.
(c) The only sub-trees that are not traversed by the DFS are those that cannot ever contain a key in range $[lo, hi]$.

5 Experiments

In this section, we present the results of experiments comparing the performance of our k-ST (for $k = 16, 32, 64$) with Snap, Ctrie, and SL, the non-blocking, randomized skip-list of the Java Foundation Classes. We used the authors' implementations of Snap and Ctrie. Java code for our k-ST is available on-line [8].

All of these data structures implement the dictionary ADT, where INSERT(key) returns FALSE if an element with this key is already in the dictionary. For SL, a RANGEQUERY is performed by executing the method `subSet(lo, true, hi, true)`, which returns a reference to an object permitting iteration over the keys in the structure, restricted to $[lo, hi]$, and then copying each of these keys into an array. This does not involve any sort of snapshot, so SL's range queries are not always linearizable. For Snap, a RANGEQUERY is performed by following the same process as SL: i.e., executing `subSet`, and copying keys into an array. However, unlike SL, iterating over the result of Snap's `subSet` causes a snapshot of the data structure to be taken. Since keys are ordered by their hashed values in Ctrie, it is hard to perform range queries efficiently. Instead, we attempt to provide an approximate lower bound on the computational difficulty of computing RANGEQUERY(lo, hi) for any derivative of Ctrie which uses the same fast snapshot technique. To do this, we simply take a snapshot, then iterate over the first $(hi-lo+1)/2$ keys in the snapshot, and copy each of these keys into an array. We explain below that $(hi - lo + 1)/2$ is the expected number of keys returned by a RANGEQUERY. To ensure a fair comparison with the other data structures, our k-ST's implementation of RANGEQUERY returns an array of keys. If it is allowed to return a list of leaves, its performance improves substantially.

Our experiments were performed on two multi-core systems. The first is a Fujitsu PRIMERGY RX600 S6 with 128GB of RAM and four Intel Xeon E7-4870 processors, each having 10 × 2.4GHz cores, supporting a total of 80 hardware threads (after enabling hyper-threading). The second is a Sun SPARC Enterprise T5240 with 32GB of RAM and two UltraSPARCT2+ processors, each having 8 × 1.2GHz cores, for a total of 128 hardware threads. On both machines, the Sun 64-bit JVM version 1.7.0_3 was run in server mode, with 512MB minimum and maximum heap sizes. We decided on 512MB after performing preliminary experiments to find a heap size which was small enough to regularly trigger garbage collection and large enough to keep standard deviations small. We also ran the full suite of experiments for both 256MB and 15GB heaps. The results were quite similar to those presented below, except that the 256MB heap caused large standard deviations, and the total absence of garbage collection with the 15GB heap slightly favoured Ctrie.

For each experiment in {5i-5d-40r-size10000, 5i-5d-40r-size100, 20i-20d-1r-size10000, 20i-20d-1r-size100}, each algorithm in {KST16, KST32, KST64, Snap, Ctrie, SL}, and each number of threads in $\{4, 8, 16, 32, 64, 128\}$, we ran 3 trials, each performing random operations on keys drawn uniformly randomly from the key range $[0, 10^6)$ for ten seconds. Operations were chosen randomly according to the experiment. Experiment "xi-yd-zr-size$s$" indicates x% probability of a randomly chosen operation to be an INSERT, y% probability of a DELETE, z% probability of RANGEQUERY($r, r + s$), where r is a key drawn uniformly randomly from $[0, 10^6)$, and the remaining $(100 - x - y - z)$% probability of a FIND. Our graphs do not include data for 1 or 2 threads, since the differences between the throughputs of all the algorithms was very small. However, we did include an extra set of trials at 40 threads for the Intel machine, since it has 40

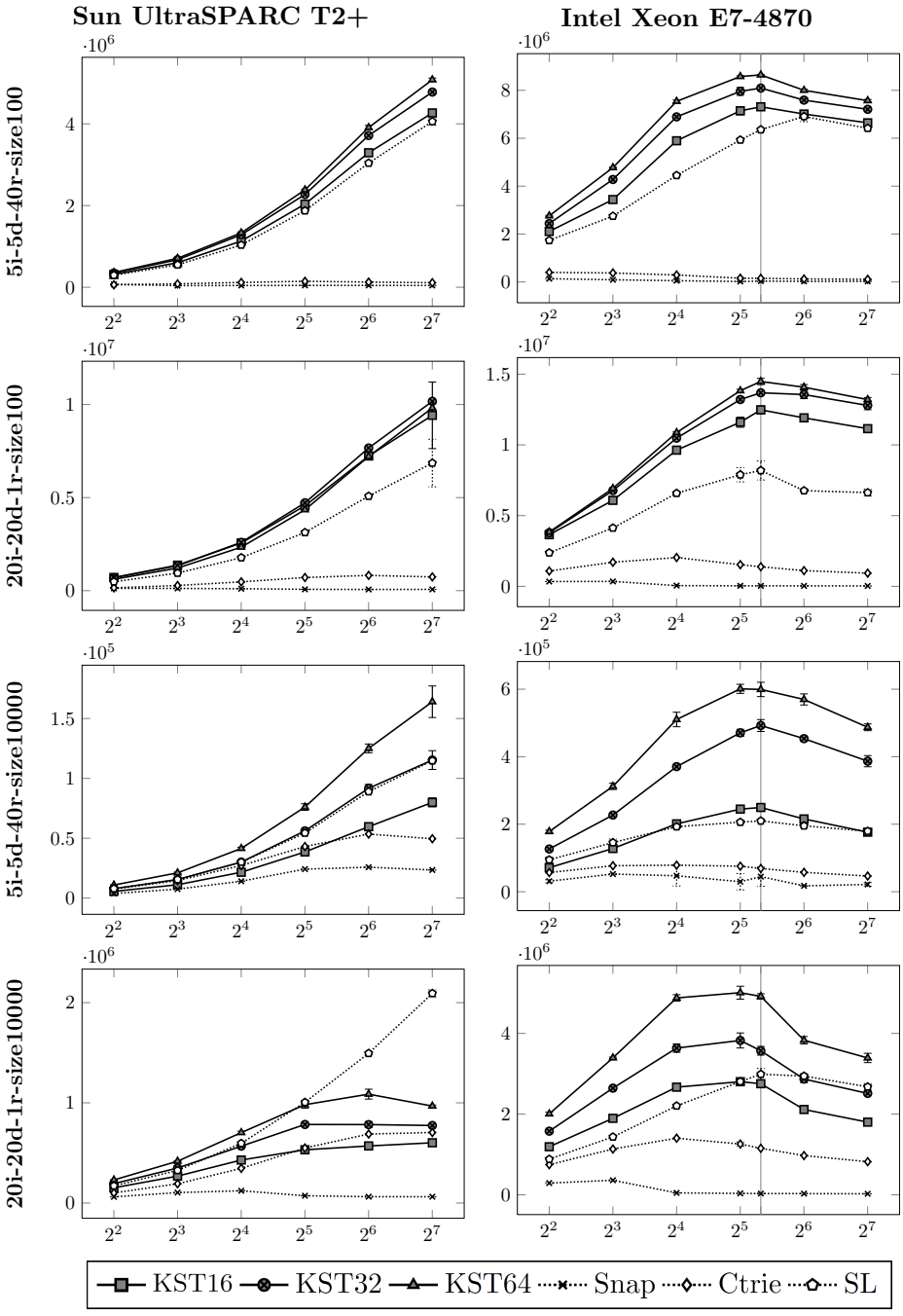

Fig. 4. Experimental results for various operation mixes (in rows) for two machines (in columns). The x-axes show the number of threads executing, and the y-axes show throughput (ops/second). The Intel machine has 40 cores (marked with a vertical bar).

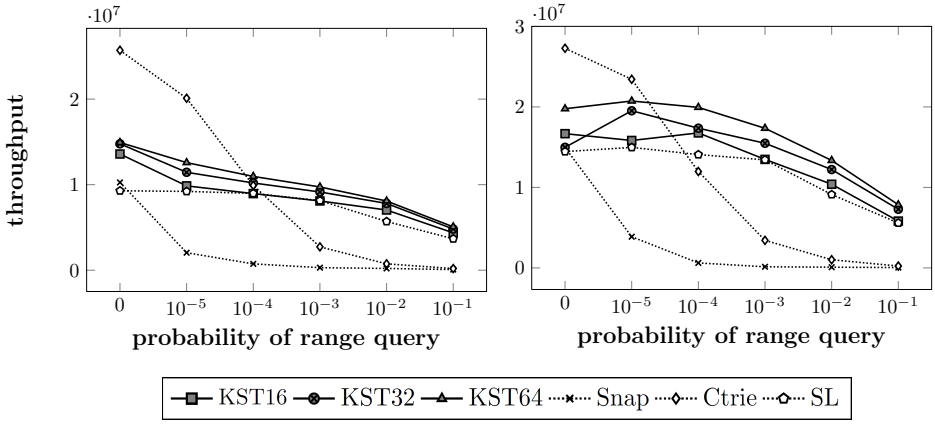

Fig. 5. Sun (left) and Intel (right) results for experiment 5i-5d-?r-100 wherein we vary the probability of range queries. Note: as we describe in Sec. 5, Ctrie is merely performing a partial snapshot, rather than a range query. The Sun machine is running 128 threads, and the Intel machine is running 80 threads.

cores. Each data structure was pre-filled before each trial by performing random INSERT and DELETE operations, each with 50% probability, until it stabilized at approximately half full (500,000 keys). Each data structure was within 5% of 500,000 keys at the beginning and end of each trial. This is expected since, for each of our experiments, at any point in time during a trial, the last update on a particular key has a 50% chance of being an INSERT, in which case it will be in the data structure, and a 50% chance of being a DELETE, in which case it will not. Thus, $(hi - lo + 1)/2$ is the expected number of keys in the data structure that are in $[lo, hi]$. In order to account for the "warm-up" time an application experiences while Java's HotSpot compiler optimizes its running code, we performed a sort of pre-compilation phase before running our experiments. During this pre-compilation phase, for each algorithm, we performed random INSERT and DELETE operations, each with 50% probability, for twenty seconds. Our experiments appear in Fig. 4. Error bars are drawn to represent one standard deviation. A vertical bar is drawn at 40 threads on the graphs for the Intel machine, marking the number of cores in the machine.

Broadly speaking, our experimental results from the Sun machine look similar to those from the Intel machine. If we ignore the results on the Intel machine for thread counts higher than 40 (the number of cores in the machine), then the shapes of the curves and relative orderings of algorithms according to performance are similar between machines. A notable exception to this is SL, which tends to perform worse, relative to the other algorithms, on the Intel machine than on the Sun machine. This is likely due to architectural differences between the two platforms. Another Intel Xeon system, a 32-core X7560, has also shown the same scaling problems for SL (see [9] technical report).

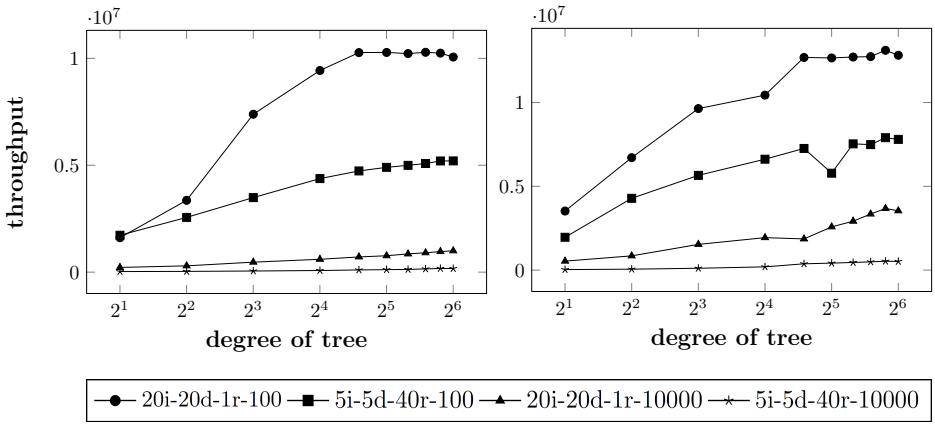

Fig. 6. Sun (left) and Intel (right) results showing the performance of the k-ST for many values of k, and for various operation mixes. The Sun machine is running 128 threads, and the Intel machine is running 80 threads.

We now discuss similarities between the experiments 5i-5d-40r-size100 and 20i-20d-1r-size100, which involve small range queries, before delving into their details. The results from these experiments are highly similar. In both experiments, all k-STs outperform Snap and Ctrie by a wide margin. Each range query causes Snap (Ctrie) to take a snapshot, forcing all updates (updates and queries) to duplicate nodes continually. Similarly, SL significantly outperforms Snap and Ctrie, but it does not exceed the performance of any k-ST algorithm. Ctrie always outperforms Snap, but the difference is often negligible. In these experiments, at each thread count, the k-ST algorithms either perform comparably, or are ordered KST16, KST32 and KST64, from lowest to highest performance.

Experiment 5i-5d-40r-size100 represents the case of few updates and many small range queries. The k-ST algorithms perform extremely well in this case. On the Sun machine (Intel machine), KST16 has 5.2 times (5.3 times) the throughput of Ctrie at four threads, and 38 times (61 times) the throughput at 128 threads. The large proportion of range queries in this case allows SL, with its extremely fast, non-linearizable RangeQuery operation, to nearly match the performance of KST16 on the Sun machine.

Experiment 20i-20d-1r-size100 represents the case of many updates and few small range queries. The k-STs are also strong performers in this case. On the Sun machine (Intel machine), KST16 has 4.7 times (3.4 times) the throughput of Ctrie at four threads, and 13 times (12 times) the throughput at 128 threads. In contrast to experiment 5i-5d-40r-size100, since there are few range queries, KST32 and KST64 do not perform significantly better than KST16. Similarly, with few range queries, the simplicity of SL's non-linearizable RangeQuery operation does not get a chance to significantly affect SL's throughput. Compared to experiment 5i-5d-40r-size100, the throughput of SL significantly decreases, relative to the k-ST algorithms. Whereas KST16 only outperforms SL by 5.2% at

128 threads on the Sun machine in experiment 5i-5d-40r-size100, it outperforms SL by 37% in experiment 20i-20d-1r-size100.

We now discuss similarities between the experiments 5i-5d-40r-size10000 and 20i-20d-1r-size10000, which involve large range queries. In these experiments, at each thread count, the k-ST algorithms are ordered KST16, KST32 and KST64, from lowest to highest performance. Since the size of its range queries is fairly large (5,000 keys), Ctrie's fast snapshot begins to pay off and, for most thread counts, its performance rivals that of KST16 or KST32 on the Sun machine. However, on the Intel machine, it does not perform nearly as well, and its throughput is significantly lower than that of SL and the k-ST algorithms. Ctrie always outperforms Snap, and often does so by a wide margin. SL performs especially well in these experiments, no doubt due to the fact that its non-linearizable RangeQuery operation is unaffected by concurrent updates.

Experiment 5i-5d-40r-size10000 represents the case of few updates and many large range queries. In this case, SL ties KST32 on the Sun machine, and KST16 on the Intel machine. However, KST64 outperforms SL by between 38% and 43% on the Sun machine, and by between 89% and 179% on the Intel machine. On the Sun machine, Ctrie's throughput is comparable to that of KST16 between 4 and 64 threads, but KST16 outperforms Ctrie by 61% at 128 threads. KST64 outperforms Ctrie by between 44% and 230% on the Sun machine, and offers between 3.1 and 10 times the performance on the Intel machine.

Experiment 20i-20d-1r-size10000 represents the case of many updates and few large range queries. On the Sun machine, SL has a considerable lead on the other algorithms, achieving throughput as much as 116% higher than that of KST64 (the next runner up). The reason for the k-ST structures' poor performance relative to SL is two-fold. First, SL's non-linearizable range queries are not affected by concurrent updates. Second, the extreme number of concurrent updates increases the chance that a range query of the k-ST will have to retry. On the Intel machine, KST64 still outperforms SL by between 21% and 136%. As in the previous experiment, Ctrie ties KST16 in throughput on the Sun machine. However, KST64 achieves 127% (270%) higher throughput than Ctrie with four threads, and 37% (410%) higher throughput at 128 threads on the Sun machine (Intel machine).

As we can see from Fig. 5, in the total absence of range queries, Ctrie outperforms the k-ST structures. However, mixing in just one range query per 10,000 operations is enough to bring it in line with the k-ST structures. As the probability of an operation being a range query increases, the performance of Ctrie decreases dramatically. Snap performs similarly to the k-ST structures in the absence of range queries, but its performance suffers heavily with even one range query per 100,000 operations.

We also include a pair of graphs in Fig. 6 for the Intel and Sun machines, respectively, which show the performance of the k-ST over many different values of k, for each of the four experiments. Results for both machines are similar, with larger values of k generally producing better results. On both machines, the curve for experiment 20i-20d-1r-size100 flattens out after $k = 24$, and 5i-5d-

40r-size100 begins to taper off after $k = 32$. Throughput continues to improve up to $k = 64$ for the other experiments. The scale of the graphs makes it difficult to see the improvement in 5i-5d-40r-size10000 but, on the Sun (Intel) machine, its throughput at $k = 64$ is 6 times (16 times) its throughput at $k = 2$. This seems to confirm our belief that larger degrees would improve performance for range queries. Surprisingly, on the Intel machine, experiment 20i-20d-1r-size10000 sees substantial throughput increases after $k = 24$. It would be interesting to see precisely when a larger k becomes detrimental for each curve.

6 Future Work and Conclusion

Presently, the k-ST structure is unbalanced, so there are pathological inputs that can yield poor performance. We are currently working on a general scheme for performing atomic, non-blocking tree updates, and we believe the results of that work will make it a simple task to design and prove the correctness of a balancing scheme for this structure.

Another issue is that, in the presence of continuous updates, range queries may starve. We may be able to mitigate this issue by having the RangeQuery operation write to shared memory, and having other updates help concurrent range queries complete. It is possible to extend the flagging and marking scheme used by the k-ST so that range queries flag nodes, and are helped by concurrent updates. While this will likely alleviate starvation in practice, it is an imperfect solution. First, it will not eliminate starvation, for the same reason that updates in the k-ST are not wait-free. More specifically, if helpers assist in flagging all of the nodes involved in a RangeQuery, they must do so in a consistent order to avoid deadlock. Moreover, the nodes of the tree do not have parent pointers, so only top-down flagging orders make sense. Therefore, it is possible to continually add elements to the end of the range and prevent the RangeQuery from terminating. Second, if range queries can be helped, we must answer questions such as how helpers should avoid duplicating work when a RangeQuery involves many nodes. Since nodes must be flagged in a consistent order by all helpers, one cannot simply split helpers up so they start flagging at different nodes. One possibility is to have helpers collaborate through a work-queue.

Despite the potential for starvation, we believe our present method of performing range queries is practical in many cases. First, range queries over small intervals involve few nodes, minimizing the opportunity for concurrent updates to interfere. Second, for many database applications, a typical workload has many more queries (over small ranges) than updates. For example, consider an airline's database of flights. Only a fraction of the queries to their database are from serious customers, and a customer may explore many different flight options and date ranges before finally purchasing a flight and updating the database.

It would be interesting to measure and compare the amount of shared memory consumed by each data structure over the duration of a trial, as well as the amount of local memory used by our range query algorithm.

In this work, we described an implementation of a linearizable, non-blocking k-ary search tree offering fast searches and range queries. Our experiments show that, under several workloads, this data structure is the only one with scalable, linearizable range queries. When compared to other leading structures, ours exhibits superior spatial locality of keys in shared memory. This makes it well suited for NUMA systems, where each cache miss is a costly mistake.

Acknowledgements. We would like to thank Faith Ellen for her extensive help in organizing and editing this paper. Our thanks also go out to the anonymous OPODIS reviewers for their helpful comments. Finally, we thank Michael L. Scott at the University of Rochester for graciously providing access to the Sun machine. This research was supported, in part, by NSERC.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sept. 1993.
2. Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, pages 65–79, 2011.
3. H. Attiya, R. Guerraoui, and E. Ruppert. Partial snapshot objects. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 336–343, New York, NY, USA, 2008. ACM.
4. G. Barnes. A method for implementing lock-free data structures. In *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.
5. A. Braginsky and E. Petrank. A lock-free b+tree. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 58–67, New York, NY, USA, 2012. ACM.
6. N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 257–268, 2010.
7. N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *PODC*, pages 6–15, 2010.
8. T. Brown and H. Avni. Range queries in non-blocking k-ary search trees. Available at `http://www.cs.toronto.edu/~tabrown/kstrq`.
9. T. Brown and J. Helga. Non-blocking k-ary search trees. In *Proc. 15th International Conference on Principles of Distributed Systems*, pages 207–211, 2011. Complete proof and code available at `http://www.cs.toronto.edu/~tabrown/ksts`, more details in Tech. Report CSE-2011-04, York University.
10. F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010. Full version in Tech. Report CSE-2010-04, York University.
11. C. Okasaki. *Purely functional data structures.* Cambridge University Press, New York, NY, USA, 1998.
12. A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *To appear in Proc. 17th ACM Symposium on Principles and Practice of Parallel Programming*, 2012.
13. N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.