Efficient Lock-Free Removing and Compaction for the Cache-Trie Data Structure

Aleksandar Prokopec[1]

Oracle Labs

Abstract. The recently proposed cache-trie data structure improves the performance of lock-free Ctries by maintaining an auxiliary data structure called a cache. The cache allows basic operations to run in expected $O(1)$, instead of the previous $O(\log n)$ bound. While earlier work showed that cache-tries improve inserts and lookups by $1.5 - 5\times$ on standard workloads, the remove operation was not previously examined. One of the main challenges of remove is to compact the trie – removing the elements should recycle the unused parts of the data structure.
In this paper, we describe a new non-compacting and two new compacting non-blocking variants of the remove operation for cache-tries. We ensure that each remove implementation runs in expected $O(1)$ time. Compared to standard Ctries, performance improvements range between 10% and 35%, depending on the size of the data structure, the parallelism level and the hardware architecture.

1 Introduction

Cache-tries [28] improve the running time of traditional lock-free hash tries [32] with a quiescently consistent auxiliary data structure called a *cache*. While cache-trie lookup and insert were shown to run in expected $O(1)$ time [26], original work on cache-tries gives almost no attention to removing elements.

This paper shows that the lock-free cache-trie remove operation also runs in expected $O(1)$ time. This operation takes care to *compact* the cache-trie – the memory management system is allowed to recycle the unused parts of the data structure. The main idea in compaction is to, after removing, speculatively detect if the affected node can be compacted, and then *freeze* it. Freezing facilitates compaction by atomically preventing subsequent updates to the candidate node.

After summarizing the earlier results and explaining how cache-tries work in Section 2, this paper brings forth the following contributions:

- A description and an implementation of a lock-free remove operation for cache-tries, both with and without compaction (Section 3).
- An optimization that brings a further $5 - 15\%$ improvement on the expected execution time (Section 3.2).
- A performance evaluation on two architectures, against three similar concurrent data structures. We find that cache-trie removes improve the execution time of standard lock-free hash tries [34] by $10 - 35\%$ and that, without compaction, removing can additionally be made $30 - 65\%$ faster (Section 4).

```
class SNode:
  val key: KeyType
  val value: ValueType
  var txn: Any

type ANode = Array<Any>
```

```
class FNode:
  val frozen: Any

val NoTxn
val FVNode
val FSNode
```

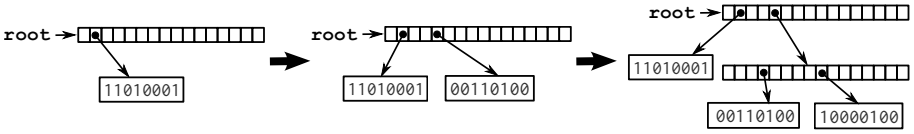
```
class CacheTrie:
  val root = new ANode(16)
  var cacheHead: Cache = nil

type Cache = Array<Any>
```

Fig. 1. Cache trie data types

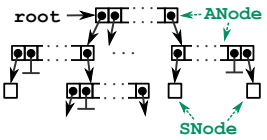
Finally, Section 5 presents the related work, and Section 6 concludes.

2 Overview of Cache-Tries

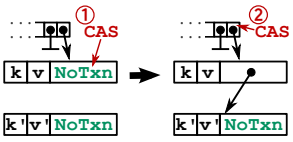
A lock-free *cache-trie* [28] is a special type of a *hash trie* data structure [3, 4, 32]. A newly created cache-trie consists of a single empty array, which has the length 16, since nodes are 16-way in our implementation[1]. Inserting a key works similar to a hash table – the 4 lowest hash code bits are used to determine the position in the table. Consider the following figure:



In the first figure above, a key with the hash code 11010001_2 occupies the index 1_{10}. The key with the hash code 00110100_2 occupies the index 4_{10} in the second figure. In the third figure, keys 00110100_2 and 10000100_2 collide at the index 4_{10}. The collision is resolved by creating another array, and using the higher hash code bits to map the keys to indices 3_{10} and 8_{10}. Collision resolution repeats recursively until running out of hash bits, and relies on a linked list thereafter.



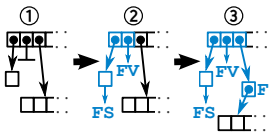
Data types. The aforementioned array nodes are modeled with the `ANode` type, shown in Figure 1. `ANode` is defined as an array of pointers of `Any` type. The `SNode` type models the leaf nodes. Each leaf holds a key and the value it is mapped to. Additionally, `SNode` objects have a mutable field `txn`, which is used by the mutating threads to announce that they are about to replace the respective `SNode`.



To insert a key into the cache-trie in a lock-free manner, a thread finds the appropriate `ANode`, and atomically replaces a `nil` array entry with a new `SNode`, using a compare-and-swap (`CAS`) instruction. A new `SNode` has the `txn` field set to a special value `NoTxn`. To replace an existing `SNode`, a mutation operation first announces the new value by `CAS`ing it into the `txn` field, as shown on the left. If the first `CAS` is successful, the corresponding array entry is replaced with a second `CAS`.

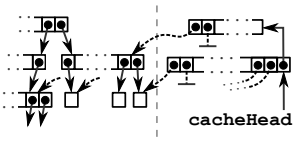
[1] Other arities are possible, but we found 16 to work well, because the node fits into a 64 byte cache-line when JVM's compressed object pointers are used.

Freezing. Mutation operations must sometimes prevent all future modifications to specific subtries. This is achieved with *freezing* [6, 23], which ensures that all subsequent `CAS` invocations on the frozen node fail.



In figure ① on the right, a thread selects an `ANode` that contains an `SNode`, a `nil` entry, and a child `ANode`. In figure ②, the thread `CAS`ed an `FSNode` value into the `txn` field of the `SNode`, and replaced `nil` with an `FVNode` value. To freeze the child `ANode`, the thread first writes an `FNode` value into the corresponding entry, as shown in Figure ③, and then recursively freezes the child. Note that, once freezing starts, it eventually completes – the reason is that a thread that observes an ongoing freeze operation will cooperatively complete the freeze. In addition, any operation that starts after the freeze started will notice the freeze if it works on the respective subtrie, and ongoing operations do only finitely many changes. The linearization point is the `CAS` that freezes the last non-frozen node in the respective subtrie.

Invariants. All operations maintain the following invariants: (1) If an `SNode` with a hash code h is reachable from the cache-trie `root`, then it is only reachable with a chain of pointers $a_0 \overset{a_0[p_0]}{\longrightarrow} [u_0 \longrightarrow] a_1 \overset{a_1[p_1]}{\longrightarrow} [u_1 \longrightarrow] \ldots \overset{a_n[p_n]}{\longrightarrow} [u_n \longrightarrow] s_{n+1}$ starting from the $a_0 =$`root`, such that $p_0 p_1 \ldots p_n$ is a prefix of h (and parts in the $[\cdot]$ brackets are optional). Here, each a_i is an `ANode`, s_i is an `SNode`, and u_i is any other type of node. (2) If a node is not reachable from the `root`, then it is frozen. (3) Once frozen, a node is not subsequently modified.



Cache. Most of the previous description applies to standard lock-free hash trie variants [31, 32, 34, 1, 2]. In these hash tries, the search time grows logarithmically with the number of keys in the hash trie. A cache-trie additionally maintains an auxiliary data structure called a *cache*, shown on the left, which speeds up the node searches. The cache is a list of C arrays, starting with the field `cacheHead`. Each array corresponds to a level ℓ, $0 \leq \ell < 4 \cdot C$, and is effectively a concatenation of the entries of the `ANode`s at level ℓ (including any missing ones). Levels are counted in multiples of 4, i.e. $0, 4, 8, 12, \ldots$ and so on. A special entry in each array contains a pointer to the next array in this list. The list is sorted, going from the largest arrays (i.e. deepest trie levels) to the smallest. The cache entries are populated lazily, so they do not always precisely match the trie. However, when they are present, they allow skipping a non-constant number of cache-trie levels.

Slow path and fast path. Consider how to implement a key lookup. The slow lookup relies on the aforementioned invariant (1) – it follows the path from the root to the unique leaf for that key. The fast path lookup attempts to first find the key in the cache, and then continues the search from some node deep in the cache-trie. If the cache entry is empty, the search reverts to the slow path from the root. Similarly, if the cache contains a frozen entry, then the respective node is potentially unreachable, and must be ignored (recall the invariants (2) and (3)). The precise pseudocode of lookup was shown in earlier work [28].

When the cache is appropriately positioned and populated, the slow path runs less frequently. When it does occasionally occur, the slow path updates the relevant cache entry, and records a *miss*. When sufficiently many misses occur, a sampling pass inspects the trie and updates the cache depth if necessary.

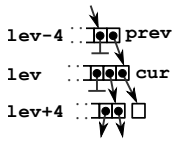
Performance. By analyzing the key distribution across levels [26], it was shown that the expected running time of the lookup and insert operations is $O(1)$. Performance evaluations on typical sizes ($\approx$1M elements) showed that cache-trie lookup performance is around $3\times$ better when compared against other hash tries, and insertion is around 33% faster [28]. Compared to the JDK 8 `ConcurrentHashMap` [14], cache-tries have faster insertion, but $1.5 - 2\times$ slower lookup.

3 Remove Operation

Depending on the implementation, a remove operation may or may not compact the cache-trie. Compaction recycles the unused parts of the data structure, and ensures that the memory footprint corresponds to the actual number of keys. For example, the JDK `ConcurrentHashMap` implementation [14] does not recycle memory, so its footprint corresponds to the maximum number of keys present at any point during its lifetime. As a benefit, removing without compaction is faster because no time is spent in housekeeping. Next, we will show one non-compacting lock-free remove operation, and two compacting variants.

3.1 Basic Implementation

The basic remove implementation does not compact the cache-trie. If the specified key exists, it is atomically removed. Otherwise, the remove operation leaves the trie intact.



Summary. We first consider the slow path version. At every step, the search is anchored at an array node `cur` at level `lev` (the parent node `prev` is at `lev-4`). The search calculates the index in `cur` and reads the respective child pointer. The child is either non-existing (`nil`), or another array node, or an `SNode`. For `nil`, the search terminates, since the key is not present, by invariant (1). For `SNode`, the search checks if the keys match, and then attempts to remove the `SNode`. Otherwise, the search resumes recursively. If the search sees a frozen node, it restarts by returning `false`.

The fast path version aims to start the search from an `ANode` within the trie, so it starts by reading the `cacheHead` field, which points to the deepest trie level. If the respective entry is not an `ANode`, this is retried at the next (higher) cache level, until reaching the root of the cache-trie.

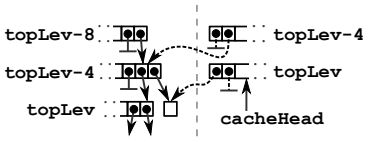
Implementation. The `remove` subroutine, shown in Figure 2, implements the slow path of the remove operation. The subroutine takes the key `k`, its hash code `h`, the current level `lev`, and the current and previous node `cur` and `prev`. The `cLev` is the cache level at which the search was entered. The subroutine returns `true` if successful, and `false` if it must be retried.

```
1 def remove(k: KeyType, h: Int, lev: Int,
2   cur: ANode, prev: ANode, cLev: Int) =
3   val pos = (h >>> lev)⊙(cur.length-1)
4   val ch = READ(cur[pos])
5   if ch == nil: return false
6   else if ch ∈ ANode:
7     return remove(k,h,lev+4,ch,cur,cLev)
8   else if ch ∈ SNode:
9     if lev ≥ cLev + 8:
10      recordMiss()
11    val txn = READ(ch.txn)
12    if txn == NoTxn:
13      if ch.key == k:
14        if CAS(ch.txn,txn,nil):
15          CAS(cur[pos],ch,nil)
16          return true
17        else:
18          return
19            remove(k,h,lev,cur,prev,cLev)
20      else: return true
21    else if txn == FSNode:
22      return false
23    else:
24      CAS(cur[pos],ch,txn)
25      return
26        remove(k,h,lev,cur,prev,cLev)
27  else:
28    return false
```

```
29 def remove(k: KeyType) =
30   val h = hash(k)
31   if ¬remove(k,h,0,root,nil,0):
32     remove(k)
33
34 def fastRemove(k: KeyType) =
35   val h = hash(k)
36   var cache = READ(cacheHead)
37   if cache == nil:
38     return remove(k)
39   val topLev =
40     trailingZeros(cache.length-1)
41   while cache ≠ nil:
42     val pos = 1+(h⊙(cache.length-2))
43     val cur = READ(cache[pos])
44     val lev =
45       trailingZeros(cache.length-1)
46     cache = READ(cache[0])
47     if cur ∈ ANode:
48       if lev < topLev-4:
49         recordMiss()
50       if remove(k,h,lev,cur,nil,lev):
51         return true
52       else:
53         continue
54     else:
55       continue
56   return remove(k)
```

Fig. 2. Remove operations

The array index `pos` is calculated in line 3, and the child `ch` is read in line 4. The `nil` and the `ANode` case are as described above. The `SNode` case first checks if the cache is misaligned – if the current trie level is sufficiently far away from cache level `cLev` (lines 9 and 10), it increments the cache miss counter by calling the `recordMiss` subroutine [28]. After that, the subroutine checks if the keys match, and performs the two-step `SNode` replacement described in Section 2.



By convention, the cache level corresponds to the level of the nodes it points to. The figure on the left shows three trie levels `topLev-8`, `topLev-4` and `topLev`. On the right, the deepest cache level is at `topLev`, and the one above is at `topLev-4`. Under this convention, the level ℓ of a cache node is the number of trailing zeros of $S-1$, where S is the cache length (note: the zeroth entry in the cache is a pointer to the next cache level). For example, the cache with $S = 17$ elements is at level 4 – note that its pointees are likewise at level 4. Separately, the index in the cache for the hash code h is determined by the ℓ lowest bits, i.e. with the expression $h \odot (S-2)$, where $\odot$ is the bitwise-and operation.

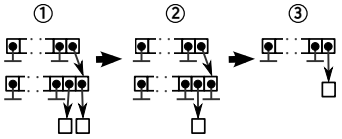
The `fastRemove` subroutine in Figure 2 iteratively traverses the cache levels until a call to `remove` in line 50 is successful. If the current cache level `lev` is less than `topLev-4`, where `topLev` is the deepest cache level, a cache miss is recorded in line 49. Note that using the cache level `topLev-4` is not a miss, because the deepest cache node can point to an `SNode`, indicating that the corresponding `ANode` must be looked up one level above, as shown in the previous figure.

```
1 def remove(k: KeyType, h: Int, lev: Int,
2   cur: ANode, prev: ANode, cLev: Int) =
  ...
6   else if old ∈ ANode:
7     val status =
8       remove(k,h,lev+4,old,cur,cLev)
9     if status ∧ prev ≠ nil:
10      if isCompactible(cur):
11        compactNode(cur,prev,h,lev)
12    return status
  ...
14        if CAS(ch.txn,txn,nil):
15          CAS(cur[pos],ch,nil)
16          if isCompactible(cur):
17            compactNode(cur,prev,h,lev)
18          return true
  ...
26        remove(k,h,lev,cur,prev,cLev)
27    else if old ∈ XNode:
28      completeCompaction(xn)
29      return false
  ...
```

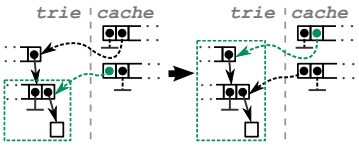
```
34 def fastRemove(k: KeyType) =
35   val h = hash(k)
36   var cache = READ(cacheHead)
37   if cache == nil:
38     return remove(k)
39   val topLev =
40     trailingZeros(cache.length-1)
41   while cache ≠ nil:
42     val pos = 1+(h⊙(cache.length-2))
43     val cur = READ(cache[pos])
44     val lev =
45       trailingZeros(cache.length-1)
  ...
50     if remove(k,h,lev,cur,nil,lev):
51       if isCompactible(cur):
52         compactUp(h,lev)
53       return true
54     else:
55       continue
56   else:
57     continue
  ...
```

Fig. 3. Adding compaction to remove operations

3.2 Cache-Trie Compaction



Summary. To compact the cache-trie, the remove operation must ensure that there is no `ANode` that has at most one `SNode` child. In the figure ① on the right, the rightmost `SNode` must be removed. Removing it in ② produces an `ANode` that has a single `SNode` child. The slow path remove has the parent pointer `prev`, so it can compact the `ANode` into its parent, as shown in ③.



However, the fast path version does not track the current node's parent, preventing it from immediately compacting. To discover the parent, the `fastRemove` operation must read the parent `ANode` from the cache nodes at the higher levels if it detects that the current node is compactible.

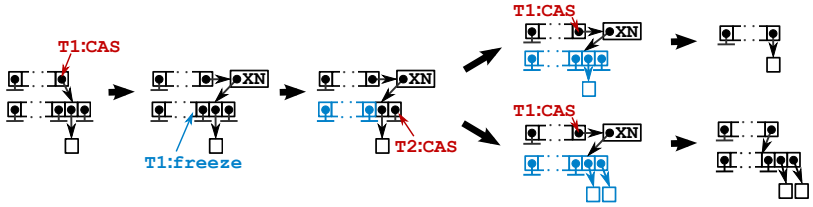
Consider the first figure on the left, in which the `remove` subroutine produced an `ANode` with a single remaining `SNode`. This `ANode` must be compacted, but the `prev` parameter is `nil`, so `remove` only "sees" the part in the green frame. In the second figure, the `fastRemove` operation reads the parent from the next level of the cache, which allows it to perform the compaction.

Implementation. Figure 3 shows the difference between the non-compacting and the compacting remove implementation. In the slow path `remove` subroutine, after successfully announcing that the `SNode` will be removed in line 14, the `isCompactible` call checks the current `ANode`, and calls `compactNode` to compact the current node if necessary. As we show briefly, compaction introduces a new node type `XNode` to mark the compact regions. If any operation observes an `XNode`, then it must first help complete the compaction, as shown in line 27.

The `fastRemove` subroutine checks if the current node is compactible as soon as its call to `remove` returns `true`. If so, the `compactUp` call in line 52 iteratively compacts the path to the root until no further compaction is possible.

Figure 4 shows the pseudocode of the different compaction operations. Compaction of a single node is done in `compactNode`, which starts by replacing the candidate node with a special `XNode` value. The `XNode` contains the pointer to the parent node `prev`, the current node `cur`, the position `ppos` of the current node in its parent, and the respective hash code and the current level. Threads that observe this node are obliged to help compaction. The `compactNode` then calls `freeze` on the candidate node to prevent further updates (freezing is described in Section 2). Finally, the current `ANode` can be replaced in its parent with the compacted version.

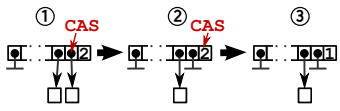
```
class XNode:
  val prev: Anode
  val ppos: Int
  val cur: ANode
  val hash: Int
  val lev: Int
```



Example. In the preceding figure, the thread T1 inserts the `XNode` and starts freezing the candidate. Before freezing completes, another thread T2 attempts to modify the candidate by inserting another `SNode`. In the first outcome, T1 succeeds in freezing the node before T2 manages to complete its update, and compaction succeeds. In the second outcome, T2 inserts the key. After freezing, T1 sees two keys in the candidate node, so it just swaps in a copy of the node.

The `compactUp` and `compactDown` subroutines are used in the fast path. These operations ascend the cache-trie on a path that corresponds to some hash code h, and invoke compaction until reaching a non-compactible node.

Counter optimization. Every successful remove operation invokes `isCompactible` to traverse all the entries in the candidate node, and check if the node can be potentially compacted. This check can be made more efficient by adding a counter into each `ANode`, which tracks the number of non-`nil` entries. This counter is updated after the linearization point, as shown in the figure above, and it is quiescently consistent – its value is guaranteed to be correct after the operations complete.



3.3 Correctness Discussion

For space reasons, we omit the precise proofs. Instead, we refer to the existing analysis with a similar structure [26], and we briefly discuss the main points.

Linearizability. To show that the remove is correct and linearizable, we identify the linearization points, and show that they preserve the invariants. Concretely, the `CAS` instruction in line 14 of `remove` is the linearization point. Other `CAS`es do not change the state, and neither of them violates the invariants.

```
1 def freeze(cur: ANode) =
2   var i = 0
3   while i < cur.length:
4     val ch = READ(cur[i])
5     if ch == nil:
6       if ¬CAS(cur[i],ch,FVNode):
7         continue
8     else if ch ∈ SNode:
9       val txn = READ(ch.txn)
10      if txn == NoTxn:
11        if ¬CAS(ch.txn,NoTxn,FSNode):
12          continue
13      else if txn ≠ FSNode:
14        CAS(cur[i],ch,txn)
15        continue
16    else if ch ∈ ANode:
17      val fn = new FNode(ch)
18      CAS(cur[i],ch,fn)
19      continue
20    else if ch ∈ FNode:
21      freeze(ch.frozen)
22    else if ch ∈ XNode:
23      completeCompaction(ch)
24      continue
25    i += 1
26
27 def isCompactible(cur: ANode) =
28   var found = nil
29   var i = 0
30   while i < cur.length:
31     val ch = READ(cur[i])
32     if ch ∈ SNode ∧ found == nil:
33       found = ch
34     else:
35       return false
36     i += 1
37   return true
38
39 def compactNode(cur: ANode,
40   prev: ANode, h: Int, lev: Int) =
41   val pmask = prev.length-1
42   val ppos = (h >>> (lev-4))⊙pmask
43   val xn =
44     new XNode(prev,ppos,cur,h,lev)
45   if CAS(prev[ppos],cur,xn):
46     return completeCompaction(xn)
47   else:
48     return false
49 def completeCompaction(xn: XNode) =
50   freeze(xn.cur)
51   var compact = nil
52   var i = 0
53   while i < xn.cur.length:
54     val ch = READ(xn.cur[i])
55     if ch ∈ SNode ∧ compact == nil:
56       compact = ch
57     else:
58       compact = createANode(xn.cur)
59       break
60     i += 1
61   if compact ∈ SNode:
62     compact = createSNode(compact)
63   CAS(xn.prev[xn.ppos],xn,compact)
64   return compact == nil ∨
65     compact ∈ SNode
66
67 def compactUp(h: Int, from: Int) =
68   var cache = READ(cacheHead)
69   while cache ≠ nil:
70     val ppos = 1+(h⊙(cache.length-2))
71     val prev = READ(cache[ppos])
72     val lev =
73       trailingZeros(cache.length-1)
74     cache = READ(cache[0])
75     if lev ≥ from ∨ prev ∈ SNode:
76       continue
77     val pos =
78       (h >>> lev)⊙(prev.length-1)
79     val cur = READ(prev[pos])
80     if cur ∉ ANode:
81       continue
82     if ¬compactDown(h,lev+4,cur,prev):
83       return
84   compactDown(h,0,root,nil)
85
86 def compactDown(h: Int, lev: Int,
87   cur: ANode, prev: ANode) =
88   val pos = (h >>> lev)⊙(cur.length-1)
89   val ch = READ(cur[pos])
90   if ch ∈ ANode:
91     if ¬compactDown(h,lev+4,ch,cur):
92       return false
93   if isCompactible(cur) ∧ prev ≠ nil:
94     if compactNode(cur,prev,h,lev):
95       return true
96   return false
```

Fig. 4. Compaction operations

Lock-freedom. We must show that, for any failed `CAS`, the trie state changes in a finite number of steps. Consider, for example, the `CAS` in line 6 of `freeze`. Failure implies either a successful `CAS` *in line 14 of* `remove`, indicating concurrent success, or that another thread froze the entry, indicating local progress.

Complexity. When there is no contention among threads, we claim that the fast path runs in expected $O(1)$. We note that the expected time spent searching for the node with the specified key is $O(1)$, by the same arguments as for the cache-trie lookup operation [26]. The only variable amount of time could be spent in the `compactUp` subroutine, whose worst-case is indeed $O(\log n)$. However, it was shown that the pair of levels with $\approx 87\%$ or more keys is expected to be at

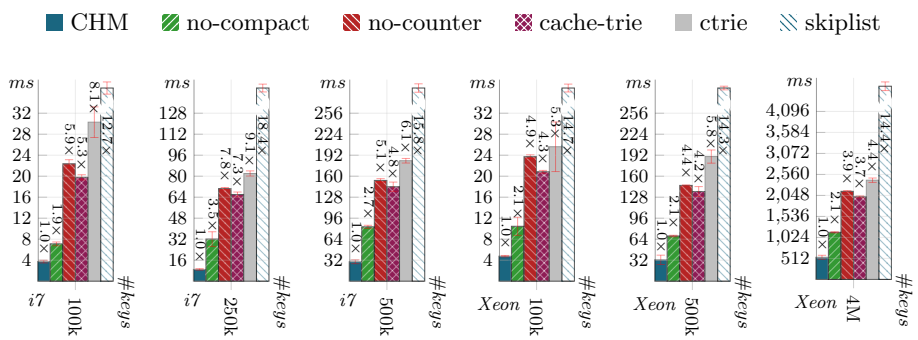


Fig. 5. Single Threaded Performance Comparison between Remove Implementations

the level of the cache [26], which is $O(1)$ levels away from the key. At that level, the expected number of entries in the `ANode` is above 2. Therefore, the number of compacted levels is expected to be constant, and `fastRemove` is $O(1)$.

4 Evaluation

We implemented cache-tries in Scala, and compared different remove implementations against similar data structures: JDK `ConcurrentHashMap` [14], Scala standard library Ctries [34], and the concurrent skip list from the JDK [43]. The single threaded benchmark takes an existing cache-trie and removes all of its N keys, where N is $100k$, $250k$, $500k$, and $4M$. The multithreaded benchmark alternates the number of concurrent threads that are removing the elements. The benchmarks were executed on two machines. The first is an Intel i7-4900MQ 3.80 GHz quad-core CPU with hyperthreading, dual-channel memory and 32GB RAM. The second machine is a dual-socket with 2 Intel Xeon E5-2683 3.00GHz tetradeca-core CPUs with hyperthreading, quad-channel memory and 32 GB RAM. We used the ScalaMeter tool to run the benchmarks [22], and we followed the standard performance evaluation techniques for the JVM [11]. We ran each benchmark 30 times, reporting the mean and the standard deviation. Our implementation is available online [30], and integrated into the Reactors framework [40, 24, 35, 27, 25].

Single threaded performance. Figure 5 shows the results of the single threaded benchmarks on i7 and Xeon. The JDK `ConcurrentHashMap` does not compact the underlying hash table, which increases its performance at the cost of memory footprint. We therefore use the `ConcurrentHashMap` as a baseline, since it is unlikely that compacting removes can achieve better performance.

We test three different cache-trie remove variants: basic removes from Section 3.1 (*no-compact*), removes with compaction from Section 3.2 (*no-counter*), and the removes with the counter optimization (*cache-trie*). *CHM*, *ctrie* and *skiplist* represent JDK concurrent hash maps, Scala Ctries and JDK concurrent skip lists, respectively. Results show that the cache-trie without compaction is $2 - 3.5\times$ slower than that `ConcurrentHashMap`. The reason for this is that the

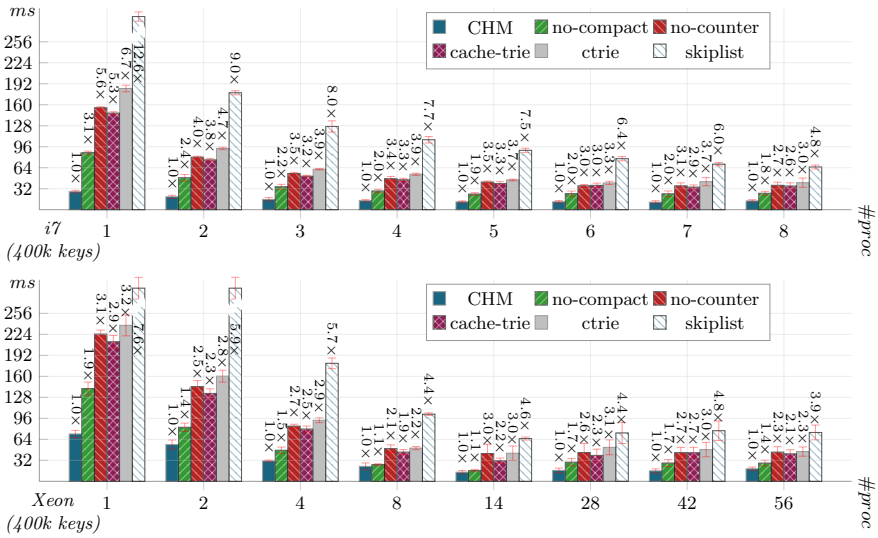


Fig. 6. Multi Threaded Performance Comparison between Remove Implementations

majority of keys are distributed across two consecutive levels of the cache [26], so the fast-path needs $\approx$ 2 pointer hops, and consequently up to $\approx$ 2 cache misses, to reach the leaf through the cache (unlike the hash table, which undergoes $\approx$ 1 pointer hop), and is consistent with earlier findings [28]. Compaction reduces performance by $1.5 - 3\times$, depending on the cache-trie size. The counter optimization from Section 3.2 improves compaction performance by only around $5 - 15\%$. This is not very surprising – the loop in the `isCompactible` subroutine (which the counters help avoid) is not particularly expensive, since (immediately after a remove) the respective node is usually already in the L1 cache.

Multi threaded performance. Figure 6 shows the results of the multi threaded benchmarks. On the i7, we vary the number of threads from 1 to 8. We test Xeon for 1, 2, 4, 8, 14, 28, 42 and 56 threads. The results are overall consistent with the single threaded benchmarks, although the performance gap is lowered at higher parallelism levels. The i7 processor, with is dual-channel memory, saturates the memory bandwidth before reaching 4 cores. The Xeon architecture saturates the bandwidth before reaching 14 cores, and exhibits a slight slowdown at higher parallelism levels. Notably, while skip lists are $\approx 7 - 13\times$ slower in the single threaded benchmarks due to a larger number of pointer hops, they scale better, and are only $\approx 4.5\times$ slower at 4 and 14 threads, respectively.

5 Related Work

Tries were proposed by Briandais [8], and later named by Fredkin [10], as a string retrieval data structure. Several authors studied the use of tries as a dictionary for arbitrary data types [16, 4, 3]. In the recent years, a non-blocking concurrent

hash trie called Ctrie, which supported lock-free insert, lookup and remove operations, was proposed by Prokopec [32, 31]. An atomic two-keys replace operation was later proposed in the context of Patricia trees [45]. Ctries were extended with non-blocking snapshots, which, along with along with high-level compiler optimizations [36, 48], enabled efficient data-parallel traversal [34, 21, 20, 33, 42, 39]. Areias and Rocha studied how to improve performance of lock-free hash tries in the context of concurrent Prolog programs, by specializing hash tries for insert operations [1, 2]. Separately, Joisha showed that non-blocking tries can be made more efficient when the delete operation is disallowed [13]. Steindorfer studied techniques for automatically deriving the hash trie with optimal tradeoffs for a given program [46, 47]. Oshman and Shavit were the first to improve complexity of the trie operations from $O(\log n)$ to $O(\log \log n)$ with SkipTries [19], and cache-tries [28, 26] were the first to lower the complexity for trie lookups and inserts to $O(1)$. The freezing technique used in cache-tries is similar to freezing used by SnapQueues [23, 41], freezing in locality-conscious lists [5], and sealing in FlowPools [38, 37, 44] and future values [12].

There are other concurrent data structures that implement the non-blocking dictionaries. Lea's `ConcurrentHashMap` [14], available in the JDK, is loosely based on Michael's lock-free hash table description [18]. `ConcurrentHashMap` has a highly efficient wait-free lookup operation, it is a flat data structure, and it avoids compaction altogether. As such, it is a good baseline for comparison against tree-like and trie data structures, which generally do compaction and suffer cache misses due to indirections. Other concurrent hash maps are due to Liu et al. [17] and Li et al. [15]. The JDK `ConcurrentSkipListMap`, compared in Section 4, is based on Pugh's concurrent skip list [43]. Other notable concurrent trees include Bronson's SnapTree (a lock-based AVL tree) [7], lock-free binary trees from Ellen et al. [9], and Braginsky's lock-free B^+-trees [6].

6 Conclusion

We described several novel non-blocking implementations of the remove operation for the cache-trie data structure. We evaluated a compacting and a non-compacting variant, and found that the overhead of compaction is around $1.5 - 3\times$, depending on the workload. However, compared to the standard Ctrie implementation [34], the compacting remove on cache-tries is $10 - 35\%$ faster.

Although the compacting remove operation exceeds the performance of Ctries, it does represent a considerable overhead. One way to alleviate these costs may be to compact lazily, i.e. trigger compaction only after removing a considerable subset of the keys. We leave these considerations for future work.

7 Data Availability Statement and Acknowledgments

The datasets and code generated during and/or analysed during the current study are available in the figshare repository [29]:
https://doi.org/10.6084/m9.figshare.6369134

References

1. Areias, M., Rocha, R.: On the correctness and efficiency of lock-free expandable tries for tabled logic programs. In: Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324. pp. 168–183. PADL 2014, Springer-Verlag New York, Inc., New York, NY, USA (2014)
2. Areias, M., Rocha, R.: A lock-free hash trie design for concurrent tabled logic programs. Int. J. Parallel Program. 44(3), 386–406 (Jun 2016)
3. Bagwell, P.: Ideal hash trees (2001)
4. Baskins, D.: The Judy array implementation. http://judy.sourceforge.net/ (2000)
5. Braginsky, A., Petrank, E.: Locality-conscious lock-free linked lists. In: Proceedings of the 12th International Conference on Distributed Computing and Networking. pp. 107–118. ICDCN'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=1946143.1946153
6. Braginsky, A., Petrank, E.: A lock-free B+tree. In: Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 58–67. SPAA '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2312005.2312016
7. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. SIGPLAN Not. 45(5), 257–268 (Jan 2010), http://doi.acm.org/10.1145/1837853.1693488
8. De La Briandais, R.: File searching using variable length keys. In: Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference. pp. 295–298. IRE-AIEE-ACM '59 (Western), ACM, New York, NY, USA (1959), http://doi.acm.org/10.1145/1457838.1457895
9. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. pp. 131–140. PODC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1835698.1835736
10. Fredkin, E.: Trie memory. Commun. ACM 3(9), 490–499 (Sep 1960), http://doi.acm.org/10.1145/367390.367400
11. Georges, A., Buytaert, D., Eeckhout, L.: Statistically Rigorous Java Performance Evaluation. SIGPLAN Not. 42(10), 57–76 (Oct 2007), http://doi.acm.org/10.1145/1297105.1297033
12. Haller, P., Prokopec, A., Miller, H., Klang, V., Kuhn, R., Jovanovic, V.: Scala improvement proposal: Futures and promises (SIP-14) (2012), http://docs.scala-lang.org/sips/pending/futures-promises.html
13. Joisha, P.G.: Sticky tries: Fast insertions, fast lookups, no deletions for large key universes. In: Proceedings of the 2014 International Symposium on Memory Management. pp. 35–46. ISMM '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2602988.2602998
14. Lea, D.: Doug Lea's workstation (2014), http://g.oswego.edu/
15. Li, X., Andersen, D.G., Kaminsky, M., Freedman, M.J.: Algorithmic improvements for fast concurrent cuckoo hashing. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 27:1–27:14. EuroSys '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2592798.2592820
16. Liang, F.M.: Word Hy-phen-a-tion by Com-pu-ter. Ph.D. thesis, Stanford University, Stanford, CA 94305 (Jun 1983), also available as Stanford University, Department of Computer Science Report No. STAN-CS-83-977

17. Liu, Y., Zhang, K., Spear, M.: Dynamic-sized nonblocking hash tables. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing. pp. 242–251. PODC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2611462.2611495
18. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures. pp. 73–82. SPAA '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/564870.564881
19. Oshman, R., Shavit, N.: The skiptrie: Low-depth concurrent search without rebalancing. In: Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing. pp. 23–32. PODC '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2484239.2484270
20. Prokopec, A., Petrashko, D., Odersky, M.: Efficient lock-free work-stealing iterators for data-parallel collections. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 248–252 (March 2015)
21. Prokopec, A.: Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime. Ph.D. thesis, IC, Lausanne (2014)
22. Prokopec, A.: Scalameter website (2014), http://scalameter.github.io
23. Prokopec, A.: SnapQueue: Lock-free queue with constant time snapshots. In: Proceedings of the 6th ACM SIGPLAN Symposium on Scala. pp. 1–12. SCALA 2015, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2774975.2774976
24. Prokopec, A.: Pluggable scheduling for the reactor programming model. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control. pp. 41–50. AGERE 2016, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/3001886.3001891
25. Prokopec, A.: Accelerating by idling: How speculative delays improve performance of message-oriented systems. In: Rivera, F.F., Pena, T.F., Cabaleiro, J.C. (eds.) Euro-Par 2017: Parallel Processing. pp. 177–191. Springer International Publishing, Cham (2017)
26. Prokopec, A.: Analysis of Concurrent Lock-Free Hash Tries with Constant-Time Operations. ArXiv e-prints (Dec 2017)
27. Prokopec, A.: Encoding the building blocks of communication. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 104–118. Onward! 2017, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3133850.3133865
28. Prokopec, A.: Cache-tries: Concurrent lock-free hash tries with constant-time operations. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '18, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3178487.3178498
29. Prokopec, A.: Efficient lock-free removing and compaction for the cache-trie data structure. https://doi.org/10.6084/m9.figshare.6369134 (2018)
30. Prokopec, A.: Reactors.io website (2018), http://reactors.io
31. Prokopec, A., Bagwell, P., Odersky, M.: Cache-Aware Lock-Free Concurrent Hash Tries. Tech. rep. (2011)
32. Prokopec, A., Bagwell, P., Odersky, M.: Lock-Free Resizeable Concurrent Tries, pp. 156–170. LCPC 2011, Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
33. Prokopec, A., Bagwell, P., Rompf, T., Odersky, M.: A generic parallel collection framework. In: Proceedings of the 17th international conference on Parallel processing - Volume Part II. pp. 136–147. Euro-Par'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dl.acm.org/citation.cfm?id=2033408.2033425

34. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent tries with efficient non-blocking snapshots. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 151–160. PPoPP '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2145816.2145836
35. Prokopec, A., Haller, P., Odersky, M.: Containers and aggregates, mutators and isolates for reactive programming. In: Proceedings of the Fifth Annual Scala Workshop. pp. 51–61. SCALA '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2637647.2637656
36. Prokopec, A., Leopoldseder, D., Duboscq, G., Würthinger, T.: Making collection operations optimal with aggressive jit compilation. In: Proceedings of the 8th ACM SIGPLAN International Symposium on Scala. pp. 29–40. SCALA 2017, ACM, New York, NY, USA (2017), http://doi.acm.org/10.1145/3136000.3136002
37. Prokopec, A., Miller, H., Haller, P., Schlatter, T., Odersky, M.: FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction, Proofs. Tech. rep. (2012)
38. Prokopec, A., Miller, H., Schlatter, T., Haller, P., Odersky, M.: Flowpools: A lock-free deterministic concurrent dataflow abstraction. In: LCPC. pp. 158–173 (2012)
39. Prokopec, A., Odersky, M.: Near optimal work-stealing tree scheduler for highly irregular data-parallel workloads. In: Cascaval, C., Montesinos, P. (eds.) Languages and Compilers for Parallel Computing. pp. 55–86. Springer International Publishing, Cham (2014)
40. Prokopec, A., Odersky, M.: Isolates, channels, and event streams for composable distributed programming. In: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!). pp. 171–182. Onward! 2015, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2814228.2814245
41. Prokopec, A., Odersky, M.: Conc-Trees for Functional and Parallel Programming, pp. 254–268. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-29778-1_16
42. Prokopec, A., Petrashko, D., Odersky, M.: On lock-free work-stealing iterators for parallel data structures p. 10 (2014)
43. Pugh, W.: Concurrent maintenance of skip lists. Tech. rep., College Park, MD, USA (1990)
44. Schlatter, T., Prokopec, A., Miller, H., Haller, P., Odersky, M.: Multi-lane flowpools: A detailed look p. 13 (2012)
45. Shafiei, N.: Non-blocking patricia tries with replace operations. In: 2013 IEEE 33rd International Conference on Distributed Computing Systems. pp. 216–225 (July 2013)
46. Steindorfer, M.J., Vinju, J.J.: Optimizing hash-array mapped tries for fast and lean immutable jvm collections. SIGPLAN Not. 50(10), 783–800 (Oct 2015), http://doi.acm.org/10.1145/2858965.2814312
47. Steindorfer, M.J., Vinju, J.J.: Towards a software product line of trie-based collections. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. pp. 168–172. GPCE 2016, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2993236.2993251
48. Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., Olukotun, K.: Composition and reuse with compiled domain-specific languages. In: Proceedings of the 27th European Conference on Object-Oriented Programming. pp. 52–78. ECOOP'13, Springer-Verlag, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-39038-8_3