Cache-Tries: Concurrent Lock-Free Hash Tries with Constant-Time Operations

Aleksandar Prokopec
Oracle Labs
aleksandar.prokopec@gmail.com

Abstract

Concurrent non-blocking hash tries have good cache locality, and horizontally scalable operations. However, operations on most existing concurrent hash tries run in $O(\log n)$ time.

In this paper, we show that the concurrent hash trie operations can run in expected constant time. We present a novel lock-free concurrent hash trie design that exerts less pressure on the memory allocator. This hash trie is augmented with a quiescently consistent cache, which permits the basic operations to run in expected $O(1)$ time. We show a statistical analysis for the constant-time bound, which, to the best of our knowledge, is the first such proof for hash tries. We also prove the safety, lock-freedom and linearizability properties. On typical workloads, our implementation demonstrates up to 5× performance improvements with respect to the previous hash trie variants.

CCS Concepts • **Theory of computation → Concurrent algorithms**; *Shared memory algorithms*; • **Computing methodologies → Concurrent algorithms**;

Keywords concurrent data structures, lock-free hash tries, constant-time hash tries, expected constant time

ACM Reference Format:
Aleksandar Prokopec. 2018. Cache-Tries: Concurrent Lock-Free Hash Tries with Constant-Time Operations. In *PPoPP '18: PPoPP '18: 23nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3178487.3178498

1 Introduction

Since the first known proposal by Briandais [11], tries have moved beyond their original string retrieval use-case. The idea of guiding the search with individual string characters was applied to hash tries, which store arbitrary key types using their hash-code bits [4, 6, 23]. The growing interest in concurrent data structures lead to the first non-blocking hash trie, named Ctrie [33], whose key advantage was improved scalability [34]. Subsequent research focused on additional non-blocking operations, such as multi-key transactions [43] and lazy snapshots [36], as well as on enhancing performance. In the context of concurrent Prolog, Areias described an insert-only variant with improved scalability [1, 2]. Joisha proposed another insert-only variant suitable for logging, Bloom filters and garbage collection [19], and Steindorfer described techniques for improving the memory footprint and cache locality [44]. However, most existing research on tries does not improve their asymptotic running time. Since anecdotal evidence suggests that lookup is a predominantly used dictionary operation [17, page 300], it remains unclear if tries should replace concurrent hash tables.

In the past, it was frequently taken for granted that concurrent hash trie operations run in $O(\log n)$ time [3, 26, 34, 36]. This assumption is possibly based on the fact that the depth of a perfectly balanced k-ary tree is $\Theta(\log n)$. However, a hash trie is not necessarily balanced and its depth can reach $O(n)$ – a trivial example involves a non-uniformly distributed hash function. Furthermore, as shown before [26], hash trie operations are not constrained by the hash trie depth, and can execute in $O(\log \log n)$ time.

In this work, we propose a novel concurrent, lock-free hash trie design that uses fewer allocations and fewer indirections compared to the original Ctrie [34]. We enhance the new hash trie variant with an auxiliary data structure called a *cache*. We call the resulting data structure a *cache-trie*. By analyzing the key depth distribution, we show that, given a universal hash function, the *expected* key depth in a hash trie (and by extension cache-trie) is $O(\log n)$. We then prove that when a cache-trie is augmented with a cache, its operations run in expected $O(1)$ time.

Concretely, we bring forth the following contributions:

- We present a novel lock-free concurrent hash trie design that uses fewer object allocations compared to the previous variants [34] (Section 3.1 and 3.2).
- We describe the *cache-trie* data structure, which extends the concurrent hash trie with a quiescently consistent cache. This allows insert, lookup and remove to run in expected $O(1)$ time (Section 3.4).

PPoPP '18, February 24–28, 2018, Vienna, Austria

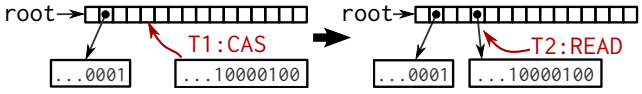
ACM ISBN 978-1-4503-4982-6/18/02…$15.00
https://doi.org/10.1145/3178487.3178498

- We present a statistical analysis of the cache-trie operations and prove the $O(1)$ time bound (Section 4.1).
- We prove that all the operations are correct, linearizable and lock-free (Section 4.2).
- We evaluate the Scala implementation of the proposed data structure against similar concurrent data structures, and show up to 5× performance improvements with respect to previous hash trie variants (Section 5).

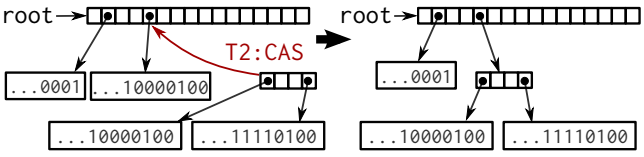
We then give a summary of related work in Section 6, and conclude in Section 7. We start with an overview that informally explains how cache-tries work.

2 Overview

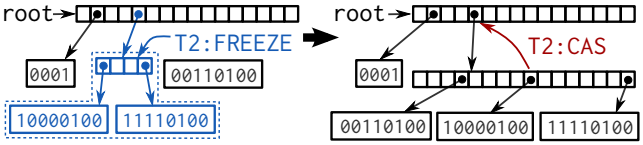
Before showing the concrete implementation, we briefly consider one specific cache-trie execution scenario. A cache-trie is identified with a `root` reference, which points to an array of pointers, called an *array node*. This array is initially empty, but gets populated with pointers to *single nodes* as keys get inserted. Consider the following cache-trie, which contains a pointer to a single node whose key has a hash code ending with $0001_2 = 1_{10}$. The position of this key is 1, as dictated by the 4 lowest bits. Assume now that some thread T1 decides to insert a key whose hash-code ends with 10000100. Thread T1 executes a CAS at the empty location $0100_2 = 4_{10}$, and after that, another thread T2 can read it.



Assume now that the other thread T2 decides to insert another key whose hash-code ends with 11110100. The location 4 of the new key coincides with the previously inserted key, so the cache-trie must be extended at the next level. T2 decides not to waste an array of length 16 on just two keys, so it creates a new array of length 4, in which the previous key occupies the position $1000_2\%100_2 = 00_2 = 0_{10}$, and the new key occupies the position $1111_2\%100_2 = 11_2 = 3_{10}$. T2 then executes a CAS at the location 4 at the first level. After the CAS completes, both keys are contained in the cache-trie:

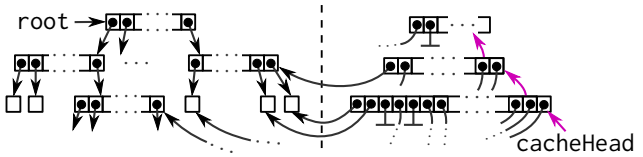


Let T2 now insert another key whose hash-code ends with 00110100. Once more, T2 encounters a collision at level 0, and then encounters another collision at the next level. Instead of immediately adding an array node at a deeper level, T2 decides to replace the narrow array node of length 4 with another array node of length 16. However, T2 cannot immediately execute a CAS at the position 4 at the root – doing so would allow a race condition in which another thread T3 modifies the narrow node after it was removed from the root array. Instead, T2 must first atomically *freeze* the narrow node to prevent further updates (details of freezing are shown later). Only after the narrow node is frozen, the pointer at the position 4 in the root can be swapped out.



Once the number of levels in the cache-trie grows sufficiently, the majority of work consists of traversing the inner nodes. The search gets progressively slower.

Cache. This is where we introduce our key contribution. We observe that pointers to nodes that are close to the leafs would likely boost the key search. We therefore extend the cache-trie with a singly-linked list of arrays. Each array holds pointers to nodes in one level of the trie, and the list is sorted from the deeper levels to the root. We call this list of arrays a *cache*. A fully populated cache of depth d has an array at each level i, $1 \leq i \leq d$, that is effectively a concatenation of all the trie nodes at level i (including any missing ones). These arrays do not perfectly match the trie because the cache is populated lazily and because parts of the trie go deeper than d. An example cache-trie is shown in the following figure:



The `cacheHead` pointer refers to the list of arrays (list pointers are violet). An array entry is either `null`, or points to a node at the respective level (each array has 16^i entries, so we omit some entries and use the "..." notation). A lookup or an insert operation starts by inspecting the *deepest* level of the cache and jumps to an inner node if a pointer exists, or otherwise reverts to a slow search from the `root`.

Challenges. There are three obstacles with this design.
(1) It is unclear how to keep the trie and the cache synchronized using single-word CAS instructions. In a non-blocking concurrent data structure, efficiently updating multiple memory locations with a single CAS is not straightforward.
(2) It is not obvious if a cache level can be selected optimally. A cache should target the level with the majority of keys (if such a level exists). At the same time, the cache must not be larger than the trie, since it is expensive to maintain.
(3) Finally, even if we knew how to pick the cache level based on the key depth distribution, it is unclear how to retrieve the distribution correctly and optimally.

The next section shows how to overcome these obstacles.

3 Design and Implementation

In this section, we present the implementation of the cache-trie data structure. We start with the basic lookup and insert operations that do not use the cache, and then explain how

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

type ANode = Array<Any>

val NoTxn

val FSNode

val FVNode

class FNode
  val frozen: Any

class ENode
  val parent: ANode
  val parentpos: Int
  val narrow: ANode
  val hash: Int
  val level: Int
  var wide: ANode

class CacheTrie
  val root = new ANode(16)
```

Figure 1. Basic Cache-Trie Data Types

```
1 def lookup(key: KeyType, hash: Int, level: Int,
2   cur: ANode): ValueType =
3   val pos = (hash >>> level)⊙(cur.length - 1)
4   val old = READ(cur[pos])
5   if (old == null ∨ old == FVNode)
6     return null
7   else if (old ∈ ANode)
8     return lookup(key, hash, level + 4, old)
9   else if (old ∈ SNode)
10    if (old.key == key) return old.value
11    else return null
12  else if (old ∈ ENode)
13    val an = old.narrow
14    return lookup(key, has, level + 4, an)
15  else if (old ∈ FNode)
16    return lookup(key, hash, level + 4, old.frozen)
17
18 def lookup(key: KeyType): ValueType =
19   lookup(key, hash(key), 0, root)
```

Figure 2. Lookup Operation

the cache improves them. We then show how to manage the cache, and discuss the remaining operations.

We use a simple pseudocode notation – data types are declared with the `class` keyword, local variables and fields are declared with the `var` keyword, and with `val` if they are constant. Methods are declared with a `def` keyword, and parameter types are denoted after the `:` sign. Pointer to any object has the `Any` type, and an array of objects of some type `T` has the `Array<T>` type. Arrays are indexed using the `[]` notation. Atomic reads, writes and compare-and-swap operations are in upper-case – `READ`, `WRITE` and `CAS`. We use standard control flow constructs – `if-else`, `while` and `return`, and `new` allocates an object and returns a pointer.

Data types. Figure 1 shows the basic cache-trie data types. During any quiescent period, a cache-trie consists only of `SNode` and `ANode` objects. An `SNode` is a leaf node – it holds a single key-value pair, a copy of the corresponding hash code, and a `txn` field that is initialized to a special value `NoTxn`. An `ANode` is an inner node, defined with the `type` keyword as an alias for an array of `Any` pointers. Cache-trie is a 16-way trie, so `ANodes` can contain up to 16 pointers. Each node occupies a specific level of the cache-trie, where levels are multiples of 4 – the root `ANode` is at level 0, its pointees are at level 4, and so on. At level ℓ, there can be up to 2^ℓ nodes.

Invariant. A cache-trie that consists only of `ANodes` and `SNodes` obeys the following invariant: if the cache-trie contains an `SNode` that stores a key whose hash-code bits are h, then this `SNode` is reachable with a chain of `ANode` pointers $a_0 \xrightarrow{a_0[p_0]} a_1 \xrightarrow{a_1[p_1]} \dots \xrightarrow{a_n[p_n]} s_{n+1}$ starting from the `root`, such that $p_0 p_1 \dots p_n$ is a prefix of h.

The rest of the node types, summarized in Table 1, exist only during an operation. `FSNode`, `FVNode` and `FNode` are special values used to prevent modifications of other nodes, as explained in Sections 3.2 and 3.3. The `ENode` is used to mark an `ANode` for expansion, and is used for insertion.

Table 1. Summary of Node Types

Name	Description
SNode	holds a key-value pair
ANode	inner node, holds pointers to other nodes
ENode	used to announce that a node must be expanded
FNode	prevents replacing an ANode entry
FSNode	prevents replacing an SNode entry
FVNode	prevents writing to an empty array entry

3.1 Lookup Operation

The goal of the `lookup` operation is to find the value associated with the specified key. If the key is not a part of the cache-trie, `lookup` must return the `null` value.

Summary. To find a value associated with a key, the `lookup` operation relies on the invariant specified in the previous section. The search works as follows – upon reaching an `ANode` at level ℓ, hash-code bits $[\ell, \ell+4\rangle$ are used as an index to select the pointer to the next level. This is repeated until reaching an empty entry or an `SNode`.

If the lookup operation encounters a special node, such as an `ENode`, `FNode`, `FSNode` or an `FVNode`, then that node contains sufficient information to determine the state of the cache-trie when this node was created. For example, an `FNode` contains the corresponding frozen `ANode`, which is used to continue the search. Thus, the lookup operation does not help a pending operation complete, and is wait-free.

Implementation. Figure 2 shows the pseudocode of the tail-recursive `lookup` subroutine, which given a `key`, its `hash`, the current `level` and the inner node `cur`, starts by extracting the relevant position bits of the hash-code. The branching factor of each `ANode` is either 4 or 16, so either 2 or 4 hash-code bits, starting from `level`, are taken to compute the

entry index pos in line 3 (⊙ denotes the bitwise-and operation). The index is used to atomically read the old value from the cur array in line 4. If old is null or FVNode (i.e. a non-modifiable version of null) in line 5, it means that the hash-code hash does not have a corresponding key in the cache-trie, so lookup returns null. If old is an ANode (line 7), lookup continues recursively from the next level. If old is an SNode (line 9), lookup checks if the SNode key is exactly equal to the desired key, and returns either the corresponding value or null. The remaining cases deal with ENodes and FNodes, and are covered in lines 12 to 16.

Most of the time, the cache-trie consists only of ANodes, SNodes and null values. Occasionally, during the execution of other operations, cache-tries can contain other types of nodes. The remaining code in lookup deals with these cases. If old is an ENode (line 12), then lookup encountered an expansion from a concurrent insert operation, as explained shortly in Section 3.2. Rather than helping complete the expansion, lookup uses the unexpanded ANode version narrow to continue the search. As shown in Section 4.2, this results in a well-defined linearization point. Finally, if old is an FNode (line 15), then lookup knows that the wrapped value is an ANode (Section 3.2), and continues recursively.

3.2 Insert Operation

Given a key and a value, the insert operation adds a new key-value pair if the cache-trie did not previously contain the key. Otherwise, the existing key-value pair is replaced.

Summary. Inserting searches the trie analogous to lookup. The goal is to find an ANode to which a new SNode can be added, or an existing SNode to be replaced. Once the ANode is found, one of the following scenarios occurs.

(1) If the corresponding ANode entry is empty, the key-value pair is atomically added. If the entry is occupied by a different key, then there is a collision.

(2) If there is a collision, and the ANode is wide, i.e. it has 16 slots, then insert creates a new ANode at the next level.

(3) Otherwise, if the ANode is narrow, i.e. it has only 4 slots, then insert assumes that expanding the ANode will resolve the collision. To correctly expand the narrow ANode, insert must first prevent subsequent updates to the narrow ANode by *freezing* it [29]. After the narrow ANode gets frozen (Section 3.3), its values are copied into the wide ANode.

(4) It is also possible that the colliding entry is occupied by the same key. In this case, the existing key-value pair is atomically replaced. For reasons that will become obvious once we introduce the cache in Section 3.4, insert uses two CAS instructions to replace an SNode – it first does a CAS on the txn field to announce the new SNode, and then another CAS on the corresponding ANode.

A case that we skip for brevity is when two different keys map to an identical hash-code. We resolve such collisions with special *list nodes*, similar to the ones used in Ctries [36].

```
1 def insert(k: KeyType, v: ValueType, h: Int,
2   lev: Int, cur: ANode, prev: ANode): Boolean =
3   val pos = (h >>> lev)⊙(cur.length - 1)
4   val old = READ(cur[pos])
5   if (old == null)
6     val sn = new SNode(h, k, v, NoTxn)
7     if (CAS(cur[pos], old, sn)) return true
8     else return insert(k, v, h, lev, cur, prev)
9   else if (old ∈ ANode)
10    return insert(k, v, h, lev + 4, old, cur)
11  else if (old ∈ SNode)
12    val txn = READ(old.txn)
13    if (txn == NoTxn)
14      if (old.key == key)
15        val sn = new SNode(h, k, v, NoTxn)
16        if (CAS(old.txn, NoTxn, sn))
17          CAS(cur[pos], old, sn)
18          return true
19        else return insert(k, v, h, lev, cur, prev)
20      else if (cur.length == 4)
21        val ppos = (h >>> (lev - 4))⊙(prev.length - 1)
22        val en = new ENode(prev, ppos, cur, h, lev)
23        if (CAS(prev[ppos], cur, en))
24          completeExpansion(en)
25          val wide = READ(en.wide)
26          return insert(k, v, h, lev, wide, prev)
27        else return insert(k, v, h, lev, cur, prev)
28      else
29        val sn = new SNode(h, k, v, NoTxn)
30        val an = createANode(old, sn, lev + 4)
31        if (CAS(old.txn, NoTxn, an))
32          CAS(cur[pos], old, an)
33          return true
34        else return insert(k, v, h, lev, cur, prev)
35    else if (txn == FSNode) return false
36    else
37      CAS(cur[pos], old, txn)
38      return insert(k, v, h, lev, cur, prev)
39  else if (old ∈ ENode) completeExpansion(old)
40  return false
41
42 def insert(k: KeyType, v: ValueType) =
43   if (!insert(k, v, hash(k), 0, root, null))
44     insert(k, v)
```

Figure 3. Insert Operation

Implementation. The insert subroutine in Figure 3 atomically reads the old value from the ANode. If old is null (line 5), then insert handles the case (1) – it creates a new SNode, and assigns it to the empty slot in line 7.

Next, if old is an SNode (line 11), then insert reads its txn value (line 12) to check if another transaction is in progress on that SNode. If txn is equal to NoTxn (line 13), then insert first checks if the SNode has the currently inserted key. If so, it proceeds according to the scenario (4) – it does a CAS on the txn field in line 16 to announce the new SNode, and then another CAS on the ANode in line 17 to commit.

If old is an SNode and does not contain the same key, then insert checks if the current ANode is narrow (line 20), according to the scenario (3). If so, insert must replace the current ANode cur with an equivalent wide ANode. A new ENode is created to communicate the intention to other threads. If the CAS that replaces cur in its parent prev succeeds in line

23, then insert managed to announce its intention, and can call the completeExpansion subroutine, shown shortly. The insertion then restarts from the same level, but using the new wide ANode that is stored in the wide field of the ENode.

Finally, if the current ANode is already wide (line 28), then insert follows the case (2) – it creates a new ANode with both the old SNode and the new one (this is done in the createANode subroutine), and attempts to replace old with the new ANode using the CAS instructions in lines 31 and 32.

Coming back to the txn field of the old SNode, if txn contains the FSNode value, then this indicates that a concurrent expansion froze that SNode and further updates are not allowed. In this case, the search needs to be restarted from the root to find the ENode that caused the freeze, so the insert subroutine returns false to communicate this. Finally, if txn contains any other value, such as an SNode or an ANode written by a concurrent insertion, the current thread simply helps complete that transaction with a CAS in line 37.

The remaining cases are as follows. Either old is an ENode, in which case insert helps complete the concurrent expansion; or old is an FVNode or an FNode, in which case insert returns false, and finds the corresponding ENode to complete the concurrent expansion. The user-facing insert subroutine checks if insertion returned false and restarts the insertion from the root if necessary.

```
1 def completeExpansion(en: ENode) =
2   freeze(en.narrow)
3   var wide = new Array<Any>(16)
4   copy(en.narrow, wide, en.level)
5   if (!CAS(en.wide, null, wide))
6     wide = READ(en.wide)
7   CAS(en.parent[en.parentpos], en, wide)
8
9 def freeze(cur: ANode) =
10  var i = 0
11  while (i < cur.length)
12    val node = READ(cur[i])
13    if (node == null)
14      if (!CAS(cur[i], node, FVNode)) i -= 1
15    else if (node ∈ SNode)
16      val txn = READ(node.txn)
17      if (txn == NoTxn)
18        if (!CAS(node.txn, NoTxn, FSNode)) i -= 1
19      else if (txn ≠ FSNode)
20        CAS(cur[i], node, txn)
21        i -= 1
22    else if (node ∈ ANode)
23      val fn = new FNode(node)
24      CAS(cur[i], node, fn)
25      i -= 1
26    else if (node ∈ FNode)
27      freeze(node.frozen)
28    else if (node ∈ ENode)
29      completeExpansion(node)
30      i -= 1
31    i += 1
```

Figure 4. Freezing and Expansion

3.3 Freezing and Expansion

After the node is frozen, it is guaranteed that no operation will ever modify that node again. A frozen node is effectively immutable, and can be read and copied without race conditions. This is important for expanding a narrow ANode.

Summary. Since all modifications to an SNode use its txn field, freezing consists of assigning the special value FSNode to txn. In the case of an ANode, freezing must prevent subsequent modifications at all the array entries. The entries are therefore disabled one-by-one. The null values are replaced with FVNode values, and child nodes are recursively frozen.

Note that, after freezing of an ANode starts, and before it ends, other threads may encounter a frozen entry, in which case they help complete the freezing, or see a regular entry, in which case they modify it. In either case, freezing eventually terminates in a lock-free manner. The linearization point is the last successful modification by another thread.

Expansion implementation. After insertion adds an ENode into the cache-trie to announce the expansion to other threads, expanding a narrow ANode into a wide one proceeds in three steps, as shown in the completeExpansion subroutine in Figure 4. First, the narrow ANode is *frozen*. The freeze subroutine, used to prevent updates to the narrow ANode, traverses the entries, and replaces each null with FVNode, each ANode with an FNode wrapper, and puts the FSNode value into the txn of each SNode. If freeze encounters nested ENodes, it completes those expansions before proceeding. For SNodes whose txn fields are not NoTxn, the pending changes are committed first.

Next, a wide ANode is allocated, and the keys of the frozen narrow ANode are sequentially transferred in the copy subroutine. The current thread then writes the wide ANode to the ENode's wide field in line 5. Finally, the wide ANode is written into the parent in line 7.

3.4 Cache Data Structure

The lookup and insert operations from Sections 3.1 and 3.2 run in expected $O(\log n)$ time, as shown in Section 4.1. To improve this, we augment the cache-trie data structure with an additional data structure, which we call a *cache*. The cache contains references to nodes close to where most of the keys reside. Reading an entry from the cache allows skipping $O(\log n)$ levels during the search. As shown in Section 4.1, this improves the expected operation running time to $O(1)$.

Summary. The cache is an array of pointers to ANodes and SNodes at a specific level. The core challenge in maintaining the cache is simultaneously removing the pointer from the array when the corresponding node is removed from the cache-trie. Without ensuring this, a thread could observe a key that had previously been removed. Using the single word CAS operations, it is difficult to efficiently update both the cache and the trie at the same time (notably, multi-word

```
1 type Cache = Array<Any>
2
3 class CacheNode
4   val parent: Array<Any>
5   val misses: Array<Int>
6
7 class CacheTrie
8   val root = new ANode(16)
9   var cacheHead: Cache = null
10
11 def createCache(level: Int, parent: Cache): Cache =
12   val cache = new Array(1 + (1 << level))
13   val misses = new Array(THROUGHPUT_FACTOR * #CPU)
14   cache[0] = new CacheNode(null, 8, misses)
15   return cache
```

Figure 5. Cache Data Types and Helper Functions

CAS constructions exist [15], but they require intermediate object allocation, more costly than a single CAS).

To ensure simultaneous eviction, the cache relies on the following two properties. First, if an SNode's txn field contains the NoTxn value, then there is a path from the root to that SNode (in other words, the cache-trie contains the key). Second, if an ANode contains at least a single entry that is not frozen, then there is a path from the root to that ANode. Therefore, as soon as an SNode contains a non-NoTxn value, or an ANode contains a frozen entry, then the respective node was likely removed, or is about to be removed.

To conclude, a pointer is entered into the cache lazily, and after the respective node is added to the trie. The cache-trie is designed so that the cache eviction is automatic – if a removed node is read from the cache, then that node is certainly in a state in which it is known that it was removed.

Data types. Figure 5 shows the basic cache trie data types. The Cache type alias is defined as an array of pointers. The first entry is a special CacheNode object, which contains the pointer `parent` to another cache level. The `parent` chain forms a list of caches, one for each level, such that the deepest level is at the head of the list. The `misses` array tracks statistics about cache misses, as explained in Section 3.6.

Implementation. A cache-trie lookup uses the subroutine `fastLookup`, shown in Figure 6. This subroutine reads the cache pointer in line 20. If the cache pointer is `null`, the normal `lookup` subroutine starts the search from the root. Otherwise, `fastLookup` traverses the cache chain starting at the deepest level. At each level, the value at the appropriate entry is read (line 26). If the value is a live SNode (i.e. `txn` is set to NoTxn), then `fastLookup` checks if the keys match or not, and returns accordingly. If the value is an ANode, then `fastLookup` first checks if the relevant entry in that ANode is not frozen, and then resumes the search. Otherwise, this process repeats with the next deepest level of the cache.

At this point, it becomes clear why `insert` from Figure 3 performs two CAS instructions to replace an SNode. The first CAS modifies the `txn` field, which is visible both in the cache and in the trie – by announcing the SNode replacement, the cache entry is effectively invalidated. The second CAS commits the change.

```
1 def lookup(k: KeyType, hash: Int, lev: Int,
2   cur: ANode, lastCachee: Any, cacheLevel: Int) =
3   if (lev == cacheLevel)
4     inhabit(cache, cur, hash, lev)
5   val pos = position(cur, hash, lev)
    ...
9   else if (old ∈ SNode)
10    if (lev ∉ [cacheLevel, cacheLevel + 4])
11      recordCacheMiss()
12    if (lev + 4 == cacheLevel)
13      inhabit(cache, old, hash, lev + 4)
14    if (old.key == key)
    ...
16      return lookup(key, hash, level + 4, old.frozen)
17
18 def fastLookup(k: KeyType): ValueType =
19   val h = hash(k)
20   var cache = READ(cacheHead)
21   if (cache == null)
22     return lookup(k, h, 0, root, null, -1)
23   val topLevel = countTrailingZeros(cache.length - 1)
24   while (cache ≠ null)
25     val pos = 1 + (h⊙(cache.length - 2))
26     val cachee = READ(cache[pos])
27     val level = countTrailingZeros(cache.length - 1)
28     if (cachee ∈ SNode)
29       val txn = READ(old.txn)
30       if (txn == NoTxn)
31         if (cachee.key == k) return cachee.value
32         else return null
33     else if (cachee ∈ ANode)
34       val cpos = (h >>> level)⊙(cachee.length - 1)
35       val old = READ(cachee[cpos])
36       if (old == FVNode ∨ old ∈ FNode) continue
37       if (old ∈ SNode)
38         if (READ(old.txn) == FSNode) continue
39       return lookup(k, h, level, cachee, level)
40     cache = cache[0].parent
41   return lookup(k, h, 0, root, null, topLevel)
```

Figure 6. Modified Lookup and the Fast Lookup Operation

3.5 Cache Housekeeping

Summary. A successful fast search does not update the cache – since the key is already in the cache, no further actions are necessary (this is especially important for fast lookups). To maintain the cache, every slow operation does two things. First, if it encounters a node at the level that corresponds to the deepest cache level, the operation inhabits the cache with that node. Second, if the operation encounters a level that is sufficiently far away from the cache, then it records a cache miss. If sufficiently many cache misses occur, then an operation considers adjusting the cache level.

Implementation. We modify the slow `lookup` operation in Figure 6. When `lookup` encounters an ANode or an SNode at the same level as the cache in lines 3 and 12, respectively, it calls the `inhabit` subroutine, which adds the node to the cache. When `lookup` encounters an SNode whose level is not

```
1 def inhabit(cache: Array[AnyRef], nv: Any,
2   hash: Int, cacheeLevel: Int) =
3   if (cache == null)
4     if (cacheeLevel >= 12)
5       cache = createCache(8, null)
6       CAS(cacheHead, null, cache)
7       inhabit(cache, nv, hash, cacheeLevel)
8   else
9     val length = cache.length
10    val cacheLevel = countTrailingZeros(length - 1)
11    if (cacheLevel == cacheeLevel)
12      val pos = 1 + (hash⊙(cache.length - 2))
13      WRITE(cache[pos], nv)
```

Figure 7. Inhabiting the Cache

ℓ_c or $\ell_c + 4$ in line 10, where ℓ_c is the level of the cache, it calls the `recordCacheMiss` subroutine (Section 3.6).

The `inhabit` subroutine in Figure 7 takes the `cache` array, the new cachee value `nv`, the hash-code `hash` of the cachee node, and the cachee level. The cache-trie does not have a cache when created, so `inhabit` first checks if a cache is necessary. If the cachee level is 12, `inhabit` initializes the cache at level 8. If the cache-trie already has a cache and the cache is at the same level as the cachee, then `inhabit` performs a `WRITE` in line 13. A `CAS` is not necessary, since the cache need not be entirely consistent.

3.6 Adjusting the Cache Level with Depth Sampling

Challenge. The performance improvement from using the cache is subject to placing it at the optimal level of the trie. To determine the optimal level, it is necessary to compute the distribution of keys across levels. This raises two challenges:
(1) On average, computing the distribution can take only a small fraction of time compared to the main operations.
(2) It is hard to obtain a sequentially consistent view of a concurrent data structure (such as the depths of all its keys).

Summary. To address the first challenge, we note that it is not necessary to have an exact key depth distribution – an approximate distribution is sufficient to decide about the cache level with high confidence. To address the second challenge, we note that the key depth distribution changes are slow. If we can estimate the distribution during a small enough time window, then the concurrent operations will not significantly change the key depth.

To estimate the key depth distribution, we rely on periodic depth sampling. When a fast operation, such as `fastLookup` from Section 3.4, fails to obtain an `SNode` from the cache, it falls back to a slow search and reports a cache miss. The thread does not immediately start sampling, but instead increases a counter. When the cache miss counter reaches a certain threshold (in our case, experimentally set to 2048), it triggers a sampling pass. The thread picks a random hash-code, uses it to traverse one path in the cache-trie, and records the number of `SNodes` at different levels of that path. The thread repeats this several times. Based on this sample, the thread then interpolates the distribution. Finally, the thread determines the most populated pair of consecutive levels, and adjusts the cache level if necessary. Importantly, in this approach, the sampling overhead is paid less often when the cache is well positioned. At the same time, sampling is triggered more frequently if the cache is not well positioned.

```
1 def recordCacheMiss() =
2   val cache = READ(cacheHead)
3   if (cache ≠ null)
4     val cn = cache[0]
5     val counterId = THREAD_ID % cn.misses.length
6     val count = READ(cn.misses[counterId])
7     if (count > MAX_MISSES)
8       WRITE(cn.misses[counterId], 0)
9       sampleAndAdjustCache(cache)
10    else WRITE(cn.misses[counterId], count + 1)
11
12 def sampleAndAdjustCache(cache: Array<Any>) =
13   val histogram = sampleSNodesLevels()
14   val best = findMostPopulatedLevel(histogram)
15   val prev = countTrailingZeros(cache.length - 1)
16   if (histogram[best] > histogram[prev] * 1.5)
17     adjustCacheLevel(best)
```

Figure 8. Recording Cache Misses and Sampling

Implementation. Figure 8 shows the `recordCacheMiss` subroutine. To decrease contention when counting the misses, the subroutine uses the `misses` array of the `CacheNode` object – the counter position is computed from the thread ID. When the counter reaches `MAX_MISSES`, a sampling pass gets triggered. The high-level pseudocode of the sampling pass is shown in the `sampleAndAdjustCache` subroutine in Figure 8. Note that neither cache miss counting, nor depth sampling, are linearizable. We found that the lack of consistency is tolerable – in the worst case, a race condition during sampling could select an incorrect level, but this is rare, and it gets corrected in the next sampling pass.

3.7 Other Operations

In the previous sections, we examined the lookup and the insert operations. For reasons of space, we did not closely examine the remove operation, which erases a key-value pair from the cache-trie. The `remove` subroutine is similar to the `insert` subroutine – it probes the cache, and then searches the cache-trie until finding an existing `SNode`. If the `SNode` key matches the specified key, the `SNode` is removed by first setting its `txn` to `null`, and then setting the pointer in the parent `ANode` to `null`. A more challenging aspect of the `remove` operation is removing empty `ANodes` – if all the entries of the `ANode` are `null`, then the `ANode` must be reclaimed. For this reason, `remove` first checks if the affected `ANode` is empty. If it is, `remove` freezes the `ANode` to prevent further updates, and replaces it in its parent. This is called *compression*, and it is similar to expansion from Section 3.3.

Concurrent maps typically provide some additional operations. For example, JDK also defines `putIfAbsent`, which

adds a key only if it was previously not in the map, and `replace`, which modifies an existing key-value pair. Since these operations work on a single `SNode`, they are straightforward modifications of the `insert` subroutine.

4 Analysis

4.1 Running Time

In this section, we show that the expected running time of the lookup, insert and remove operations is $O(1)$. The proof relies on establishing the key depth distribution. This leads to the expected key depth, and the expected depth of the cache, and consequently their expected distance. For reasons of space, we only list the main theorems in the main part of the paper, and keep the complete proofs in the respective technical report [31].

We define the depth d of a node in the cache-trie as $d = \ell/4$, where ℓ is the node level. We say that a key occupies depth d if its `SNode` is at the depth d.

Theorem 4.1. *Given a universal hash function, and a cache-trie that contains $n+1$ keys, the probability that an arbitrary key occupies a position at depth d is:*

$$p(d,n) = (1 - 16^{-d-1})^n - (1 - 16^{-d})^n \tag{1}$$

Proof sketch. Consider a particular key with a hash-code h at the level ℓ. Each of the hash-code bits of the other keys is chosen independently at random. By counting the events in which at least one other key has the same ℓ-prefix of the hash-code h, but not the same $(\ell+4)$-prefix, we get:

$$p(\ell,n) = \sum_{k=1}^{n} \binom{n}{k} \left(\frac{1}{16^{\ell/4}} \cdot \frac{15}{16}\right)^k \left(1 - \frac{1}{16^{\ell/4}}\right)^{n-k} \tag{2}$$

By simplifying this sum, the claim follows. □

The function $p(d,n)$ defines a family of depth probability distributions, each for a specific choice of n. Next, we show that for sufficiently large n, most of the keys are expected to occupy some pair of consecutive levels.

Theorem 4.2. *When the number of keys n tends to infinity, there exists a consecutive pair of levels for which the probability that a key occupies them is in the interval $\langle 0.8745, 0.9746\rangle$.*

Proof sketch. We define $\eta(d,n) = p(d,n) + p(d,n+1)$ and $\mu(n) = \max_d \eta(d,n)$. We then consider what happens when $n \to \infty$. For the upper bound, we require $\frac{\partial \mu(n)}{\partial n} = 0$. For the lower bound, we require $\eta(d,n) = \eta(d+1,n)$. □

Note that the cache will target exactly the pair of levels that contains the most keys. We now know that the cache will cover a large percentage of keys. We still need to prove that this level is a constant number of levels away from an average key, so we determine the expected key depth next.

Theorem 4.3. *In a cache-trie that contains n keys, the expected key depth is $E[d](n) = \log_{16} n + O(1)$.*

Proof sketch. From the definition of expectation. □

Having the expected key depth, we can establish the connection to the cache depth.

Theorem 4.4. *The expected distance between the cache depth and the key depth is $O(1)$.*

Proof sketch. We consider the probability that a key is at $d_1 = \log_{16}(n) - 2$ and $d_2 = \log_{16} n$, and conclude that the most populated pair of consecutive depths must start between d_1 and d_2. We know the expected key depth $E[d](n)$ is $O(1)$ steps away from d_1 and d_2, so the claim follows. □

4.2 Safety, Linearizability and Lock-Freedom

We first assume that the cache-trie does not use the cache, and show that the operations are consistent with the semantics of an abstract set. This helps us identify the `CAS` instructions that are the linearization points, and establish that the operations are linearizable. We then show that the operations are lock-free. Finally, we examine the fast variants of the operations, which use the cache, and show that they are safe, linearizable and lock-free. We include the complete proofs in the technical report [31].

Definition 4.5 (Consistency)**.** Relation $hasKey(n,k)$ holds if n is an `SNode` that contains k, or if n is an `ANode` at level ℓ whose entry $i = hash(k)>>\ell \bmod length(an)$ contains m such that $hasKey(m,k)$. A cache-trie is consistent with an abstract set $\mathbb{A}$ iff $k \in \mathbb{A} \Leftrightarrow hasKey(\texttt{root}, k)$.

Lemma 4.6 (Presence)**.** *If a thread reads a single node* `sn` *at time t_0, then $hasKey($*`root.sn.key`*$)$ holds at t_0.*

Lemma 4.7 (Absence)**.** *If a thread searches for k and reads* `null` *from an array node at time t_0, or if a thread reads a node* `sn` *at t_0 such that* `sn.key` $\neq k$*, then $hasKey($*`root`$,k)$ *does not hold at t_0.*

Lemma 4.8. `CAS` *instructions in lines 17, 32, and 37 (Figure 3), and in line 20 (Figure 4), change the abstract set that the cache-trie is consistent with. The rest of the* `CAS`*es do not change the corresponding abstract set.*

Theorem 4.9 (Safety)**.** *At all times t, a cache-trie is consistent with some abstract set. Cache-trie operations are always consistent with abstract set operations.*

Theorem 4.10. *Cache-trie operations are linearizable.*

Lemma 4.11. *There is a finite number of execution steps between any two cache-trie modifications, and there can only be finitely many consecutive modifications that do not change the corresponding abstract set.*

Theorem 4.12. *Cache-trie operations are lock-free.*

Theorem 4.13 (Cache Safety)**.** *Both fast insert and fast lookup are consistent with the abstract set semantics. Moreover, fast insert and fast lookup are linearizable and lock-free.*

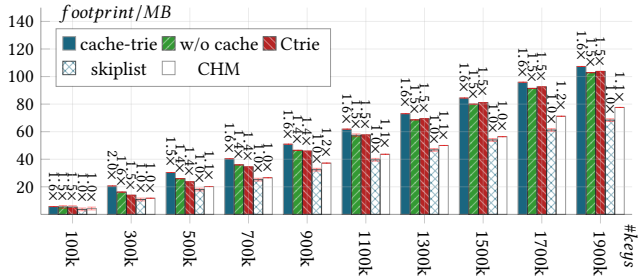


Figure 9. Memory Footprint Comparison

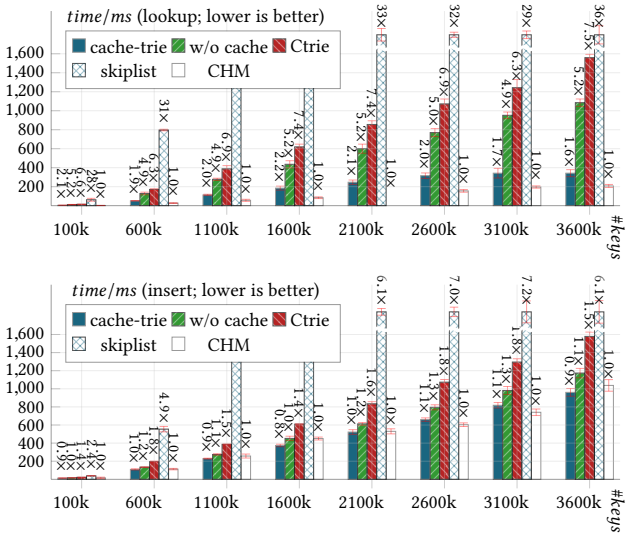


Figure 10. Single-Threaded Lookup and Insert

5 Evaluation

In this section, we compare the memory footprint and the running time of cache-tries against the Ctrie implementation from the Scala standard library [36], as well as the JDK 8 `ConcurrentHashMap` and `ConcurrentSkipListMap`. We also compare against a cache-trie variant that does not use the cache (abbreviated *w/o cache*). `ConcurrentHashMap` is a concurrent dictionary with the fastest lookup implementation, so we use it as a baseline.

We run the benchmarks on a 3.9 GHz Intel i7-4930MX quad-core processor with hyperthreading, and with frequency scaling turned off. We use the 4.10.0-32 Linux kernel and Oracle JDK 8, and we set the heap size to 6 Gb.

We use standard performance evaluation methodologies for the JVM [14], and rely on the ScalaMeter tool to conduct the benchmarks [28]. We repeat each benchmark in a new JVM process until warmup (we detect the warmup when the coefficient of variance drops below a threshold). We then repeat the benchmark 5 times, and record the execution times. This is repeated 6 times in separate JVM processes, and average value and standard deviation are reported.

Memory footprint. We start by comparing the memory footprints of various concurrent data structures in Figure 9, which shows the number of keys on the x-axis, and the footprint on the y-axis. Skip lists consume the least memory, so the multipliers above the bars are normalized against them. Since the cache duplicates a lot of pointers in the cache-trie, we would expect a significant increase. However, it turns out that the increase compared to the cache-less variant is small, and typically below 10%.

In the corresponding technical report [31], we show that most of the keys (> 87%) occupy some two consecutive levels ℓ and $\ell + 1$. Cache targets level ℓ, and contains only ~10% of those keys in the best case, or ~ 79% of keys in the worst case. On top of that, the relative overhead of a cache pointer is small. The increase from those 10% – 79% has to be divided by ~5, because the overhead of a cache pointer is only 1 word, and the footprint of an SNode is 5 words (object header, key and value pointers, hash code and the `txn` field).

Cache-tries themselves have a footprint almost identical to Ctries, and they both take around 50% more memory compared to hash tables.

Single-threaded running time. Next, we show the single-threaded benchmark results in Figure 10. The lookup benchmark receives a data structure that already contains N keys (shown on the x-axis), and then looks up each of the keys once. The respective running time is shown on the y-axis. We can see that cache-tries outperform Ctries, which are up to 7.5× slower than concurrent hash tables. Skip lists have the worst performance – they are up to 36× slower due to a large number of cache misses caused by pointer hops. Cache-trie lookups themselves are 1.6 – 2.1× slower than CHM – the source of the overhead is the extra pointer hop that a cache-trie undergoes after reading the cache (most keys are distributed across *two* consecutive levels, by Theorem 4.2).

The insert benchmark starts with an empty data structure, and sequentially inserts N different keys. For the chosen N, cache-tries are sometimes up to 20% faster than CHM, and sometimes 10% slower. The cache-less variant is within a margin of only 20% – for these sizes, allocating and replacing the nodes takes longer than traversing the trie. Ctries are around 50% slower, and skip lists are 6× slower than CHM.

Multi-threaded running time. In Figure 11, we repeat the insertion benchmark with multiple threads, and for the number of keys $50k$, $200k$ and $600k$. The threads insert the same set of keys, in the same order, so we expect a high contention. For $N = 50k$, cache-tries outperform CHM by 10% when there are 4 or less threads, and perform equally for more than 4 threads. At $N = 200k$ and $N = 600k$, cache-tries are 10 – 30% slower. The slowdown is due to a higher frequency of restarts on the slow path.

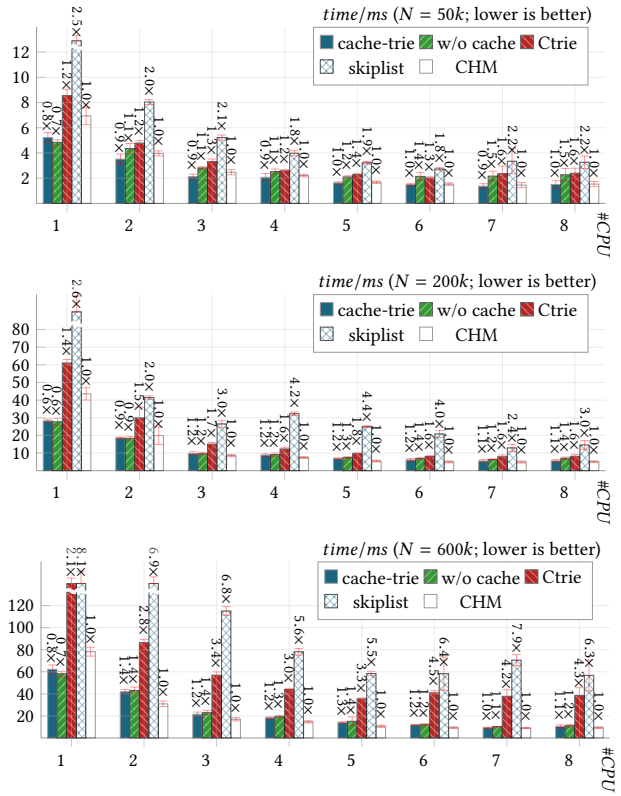


Figure 11. Multi-Threaded Insert With High Contention

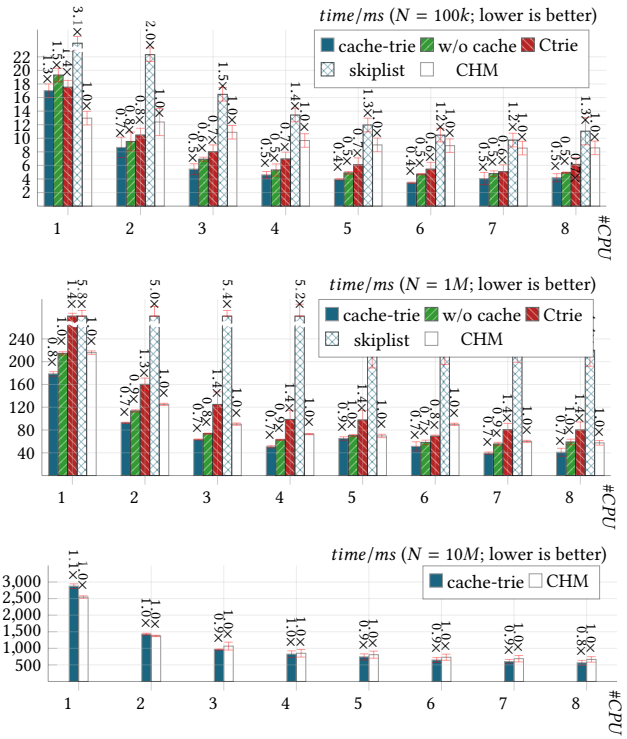


Figure 12. Multi-Threaded Insert With Low Contention

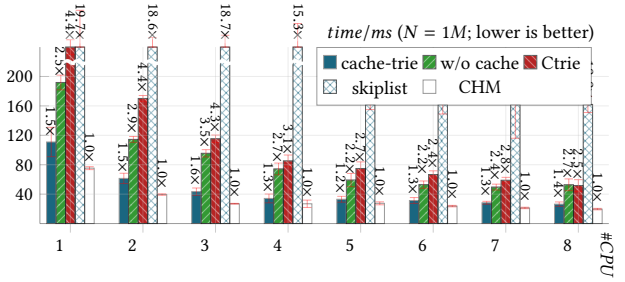


Figure 13. Multi-Threaded Lookup

Figure 12 also shows multi-threaded insertion, but this time threads operate on disjoint sets of keys. Here, cache-tries consistently outperform CHM by 30 – 50% for 100k and 1M keys, and are up to 20% faster for 10M keys.

Finally, Figure 13 shows the multi-threaded version of the lookup benchmark, for 1M keys. For the same reasons as in Figure 10, cache-tries lookups are up to 60% slower compared to CHM. Both cache-tries and CHM are much faster than all other examined data structures.

6 Related Work

The trie data structure was proposed by Briandais [11] (and later named by Fredkin [13]) as a way to efficiently store and retrieve strings. The idea of guiding the trie search with hash-code strings, and using the resulting data structure to implement the dictionary data type, was mentioned in various forms by at least several authors [4, 6, 23]. One advantage of hash tries (not to be confused with *hash trees* [24]) is to implement immutable dictionaries, used in functional languages such as Scala, Haskell or Clojure. A non-blocking concurrent hash trie, called *Ctrie*, was first proposed by Prokopec et al. [34] and proven correct [33] – in this design, the keys are stored in external nodes, and fixed-size hash-code segments are stored in internal nodes with a branching factor of 32. In Ctries, synchronization between concurrent operations on different trie levels is resolved by introducing special I-nodes, which remain in the cache-trie even when the adjacent level is modified. This improves concurrency, but also doubles the number of pointer hops and increases the average number of cache misses. Cache-tries eliminate the need for I-nodes in two ways: first, each inner node has only two possible sizes – 4 and 16. Second, when an inner node needs to be removed from the trie, its modifiable locations are set to special non-writable values – this is similar to the freezing technique used in SnapQueues [29, 40], freezing in locality-conscious lists [7], and sealing in FlowPools [37, 38].

Subsequent research on lock-free tries took several directions. One of the goals was to provide other non-blocking atomic operations, such as the atomic two-keys replace operation, first proposed in the context of Patricia trees [43].

Ctries were extended with atomic lazy $O(1)$ time snapshots [36], which allowed implementing linearizable iterators, and enabled linearizable data-parallel operations [27, 35, 41].

On a separate front, researchers aimed to improve performance by specializing operations for specific use-cases. Concurrent tries were found to be useful for tabling in concurrent Prolog, where tries store the results of redundant subcomputations. Areias et al. describe several insert-only implementations [1, 2], and show that they are more efficient and scalable compared to the general counterparts. Separately, a variant of non-blocking tries aimed at logging, Bloom filters, and implementing the GC store buffer, was shown to have a better performance when delete operations are disallowed [19]. Steindorfer describes a data structure specialization technique that automatically picks the optimal hash trie tradeoffs for a given program [45].

Other research on hash tries focused on improving their performance in general. In the context of PTries [18], the authors noted that restructuring the bit string used for the keys can sometimes improve the running time by a constant factor. Steindorfer, describes the CHAMP data structure [44], which improves the memory footprint of standard immutable hash tries with compression, and at the same time improves cache locality, but retains the same $O(\log n)$ bound. Tries with $O(\log \log n)$ running time bounds were proposed as part of the SkipTrie data structure [26], which can be viewed as a hash trie. To the best of our knowledge, $O(\log \log n)$ is the lowest previously known bound, and our work is the first concurrent trie design with a proven expected $O(1)$ running time bound. In this work, we showed the $O(\log n)$ expected hash trie key depth. Aside from the previous analysis of the b-trie depth [20], we are unaware of any prior work on depth bounds in concurrent hash tries.

Besides concurrent hash tries, many other concurrent data structures were proposed in the past. Most high-level languages expose a concurrent dictionary implementation – Java's `ConcurrentHashMap` (CHM) initially had a lock-based implementation, which was replaced in JDK 8 with a closed-addressing concurrent hash table [22], similar to Maged's original proposal [25]. The JDK 8 CHM also has a highly optimized, wait-free lookup operation. Furthermore, the JDK 8 CHM uses a parallel resize operation, triggered by insertion operations when they run out of space (this is unlike its pre-JDK 8 counterpart, which suffered scalability issues in insertions). In terms of lookup, the CHM is the most efficient and scalable concurrent dictionary that we are aware of. Despite the fact that CHM is a flat data structure that does not support fast linearizable snapshots (which makes a direct comparison with tree-like data structures unfair), we decided to use it as a baseline in our benchmarks.

Click showed an alternative open-addresing implementation [10], but not as efficient as the JDK 8 hash table (original Ctries were shown more scalable [36]). Java also exposes `ConcurrentSkipListMap`, a concurrent lock-free skip list implementation [16, 42]. One of the first concurrent binary search trees was described by Kung [21], and a lock-based concurrent AVL-tree is described by Bronson et al. [9]. The first lock-free binary search tree variant is described by Ellen et al. [12], and another is described by Braginsky and Petrank [8]. Recently, a dictionary design optimized for wait-free linearizable scan operations, similar to a B^+-tree, was proposed by Basin et al. [5].

A good overview of linearizability and lock-freedom, as well as concurrent data structures, and other concurrency-related topics is given by Herlihy and Shavit [17].

7 Conclusion

We described a novel concurrent hash trie design, and extended it with an auxiliary data structure called a cache. We showed that the resulting data structure remains linearizable in the presence of the cache, and that it is lock-free. Furthermore, we proved that the expected running time of all operations is bound by $O(1)$, and we empirically showed that cache-tries improve the performance of many previously available data structures. On typical dataset sizes, lookups are improved up to 5×, and inserts are improved up to 1.5× compared to Ctries [34]. Due to a vastly lower number of pointer hops in a typical operation, skip list lookups are outperformed by a factor of $15-22\times$, and skip list insertions by a factor of 6×. It seems likely that cache-tries outperform most other tree-based concurrent data structures for the same reasons. Moreover, unlike hash tables, cache-tries do not require resizing a large underlying array, and this usually gives rise to better scalability. In particular, cache-tries outperform concurrent hash maps by a factor of $1.1-2.5\times$ with uncontended parallel insertions, but exhibit an overhead of up to 1.8× when insertions are contended.

Based on this, can cache-tries completely replace concurrent hash tables? While the operations of both data structures run in expected $O(1)$ time, we observed that our lookup is consistently $1.2-2.0\times$ slower. The reason for this is that a cache-trie typically needs two pointer hops to find the key (as implied by Theorem 4.2), whereas a hash table needs only one. Also, since cache-tries have about 50% higher memory footprint compared to concurrent hash tables, their working set spills out of the cache more frequently, which results in a higher cache miss rate. While this overhead may be justified in applications where lookups are not of paramount importance, a further investigation is necessary to see if the cache can achieve better locality, e.g. by directly caching two levels of the cache at once. Another deciding factor is the availability of an efficient linearizable snapshot operation, which is not available for concurrent hash tables – it remains to show the exact snapshot implementation for cache-tries. We plan to investigate these questions in future work.

References

[1] Miguel Areias and Ricardo Rocha. 2014. 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 (PADL 2014)*. Springer-Verlag New York, Inc., New York, NY, USA, 168–183. https://doi.org/10.1007/978-3-319-04132-2_12

[2] Miguel Areias and Ricardo Rocha. 2016. A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs. *Int. J. Parallel Program.* 44, 3 (June 2016), 386–406. https://doi.org/10.1007/s10766-014-0346-1

[3] Phil Bagwell. 2000. *Fast And Space Efficient Trie Searches.* Technical Report.

[4] Phil Bagwell. 2001. Ideal Hash Trees. (2001).

[5] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A Key-Value Map for Scalable Real-Time Analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 357–369. https://doi.org/10.1145/3018743.3018761

[6] Douglas Baskins. 2000. The Judy Array Implementation. http://judy.sourceforge.net/. (2000).

[7] Anastasia Braginsky and Erez Petrank. 2011. Locality-conscious Lock-free Linked Lists. In *Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN'11)*. Springer-Verlag, Berlin, Heidelberg, 107–118. http://dl.acm.org/citation.cfm?id=1946143.1946153

[8] Anastasia Braginsky and Erez Petrank. 2012. A Lock-free B+Tree. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. ACM, New York, NY, USA, 58–67. https://doi.org/10.1145/2312005.2312016

[9] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. *SIGPLAN Not.* 45, 5 (Jan. 2010), 257–268. https://doi.org/10.1145/1837853.1693488

[10] Cliff Click. 2007. Towards a Scalable Non-Blocking Coding Style. (2007). http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf

[11] Rene De La Briandais. 1959. File Searching Using Variable Length Keys. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference (IRE-AIEE-ACM '59 (Western))*. ACM, New York, NY, USA, 295–298. https://doi.org/10.1145/1457838.1457895

[12] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '10)*. ACM, New York, NY, USA, 131–140. https://doi.org/10.1145/1835698.1835736

[13] Edward Fredkin. 1960. Trie Memory. *Commun. ACM* 3, 9 (Sept. 1960), 490–499. https://doi.org/10.1145/367390.367400

[14] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.* 42, 10 (Oct. 2007), 57–76. https://doi.org/10.1145/1297105.1297033

[15] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, London, UK, UK, 265–279. http://dl.acm.org/citation.cfm?id=645959.676137

[16] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2006. A Provably Correct Scalable Concurrent Skip List. (2006).

[17] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming.* Morgan Kaufmann Inc., San Francisco, CA, USA.

[18] P.G. Jensen, K.G. Larsen, and J. Srba. 2017. PTrie: Data Structure for Compressing and Storing Sets via Prefix Sharing. In *Proceedings of the 14th International Colloquium on Theoretical Aspects of Computing (ICTAC'17) (LNCS)*. Springer, 1–18. To appear.

[19] Pramod G. Joisha. 2014. Sticky Tries: Fast Insertions, Fast Lookups, No Deletions for Large Key Universes. In *Proceedings of the 2014 International Symposium on Memory Management (ISMM '14)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/2602988.2602998

[20] Charles Knessl and Wojciech Szpankowski. 2000. A Note on the Asymptotic Behavior of the Heights in b-Tries for b Large. *Electr. J. Comb.* 7 (2000). http://www.combinatorics.org/Volume_7/Abstracts/v7i1r39.html

[21] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382. https://doi.org/10.1145/320613.320619

[22] Doug Lea. 2014. Doug Lea's Workstation. (2014). http://g.oswego.edu/

[23] Franklin Mark Liang. 1983. *Word Hy-phen-a-tion by Com-pu-ter.* Ph.D. Dissertation. Stanford University, Stanford, CA 94305. Also available as Stanford University, Department of Computer Science Report No. STAN-CS-83-977.

[24] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87)*. Springer-Verlag, London, UK, UK, 369–378. http://dl.acm.org/citation.cfm?id=646752.704751

[25] Maged M. Michael. 2002. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881

[26] Rotem Oshman and Nir Shavit. 2013. The SkipTrie: Low-depth Concurrent Search Without Rebalancing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2484239.2484270

[27] Aleksandar Prokopec. 2014. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime.* Ph.D. Dissertation. IC, Lausanne. https://doi.org/10.5075/epfl-thesis-6264

[28] Aleksandar Prokopec. 2014. ScalaMeter Website. (2014). http://scalameter.github.io

[29] Aleksandar Prokopec. 2015. SnapQueue: Lock-free Queue with Constant Time Snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala (SCALA 2015)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2774975.2774976

[30] Aleksandar Prokopec. 2016. Pluggable Scheduling for the Reactor Programming Model. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/3001886.3001891

[31] Aleksandar Prokopec. 2017. Analysis of Concurrent Lock-Free Hash Tries with Constant-Time Operations. *ArXiv e-prints* (Dec. 2017). arXiv:cs.DS/1712.09636

[32] Aleksandar Prokopec. 2017. 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 (Onward! 2017)*. ACM, New York, NY, USA, 104–118. https://doi.org/10.1145/3133850.3133865

[33] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Cache-Aware Lock-Free Concurrent Hash Tries.* Technical Report.

[34] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Lock-Free Resizeable Concurrent Tries.* Springer Berlin Heidelberg, Berlin, Heidelberg, 156–170. https://doi.org/10.1007/978-3-642-36036-7_11

[35] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A Generic Parallel Collection Framework. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II (Euro-Par'11)*. Springer-Verlag, Berlin, Heidelberg, 136–147. http://dl.acm.org/citation.cfm?id=2033408.2033425

[36] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. (2012), 151–160. https://doi.org/10.1145/2145816.2145836

[37] Aleksandar Prokopec, Heather Miller, Philipp Haller, Tobias Schlatter, and Martin Odersky. 2012. *FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction, Proofs*. Technical Report.

[38] Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. 2012. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction. In *LCPC*. 158–173.

[39] Aleksandar Prokopec and Martin Odersky. 2015. 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!) (Onward! 2015)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/2814228.2814245

[40] Aleksandar Prokopec and Martin Odersky. 2016. *Conc-Trees for Functional and Parallel Programming*. Springer International Publishing, Cham, 254–268. https://doi.org/10.1007/978-3-319-29778-1_16

[41] A. Prokopec, D. Petrashko, and M. Odersky. 2015. Efficient Lock-Free Work-Stealing Iterators for Data-Parallel Collections. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 248–252. https://doi.org/10.1109/PDP.2015.65

[42] William Pugh. 1990. *Concurrent Maintenance of Skip Lists*. Technical Report. College Park, MD, USA.

[43] N. Shafiei. 2013. Non-blocking Patricia Tries with Replace Operations. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*. 216–225. https://doi.org/10.1109/ICDCS.2013.43

[44] Michael J. Steindorfer and Jurgen J. Vinju. 2015. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. *SIGPLAN Not.* 50, 10 (Oct. 2015), 783–800. https://doi.org/10.1145/2858965.2814312

[45] Michael J. Steindorfer and Jurgen J. Vinju. 2016. Towards a Software Product Line of Trie-based Collections. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2016)*. ACM, New York, NY, USA, 168–172. https://doi.org/10.1145/2993236.2993251

A Artifact appendix

A.1 Abstract

This artifact description explains how to obtain the Scala source code of the Cache Trie data structure, which is proposed in the corresponding paper, as well as a set of runnable benchmarks. The abstract contains instructions on how to run these benchmarks. The expected result is to reproduce the performance measurements from the paper. To run the benchmarks, a quad-core Intel x86 processor with hyperthreading (or better) is required. We advise using a Linux operating system with the Oracle JDK 8 runtime installed (instructions included).

A.2 Description

A.2.1 How delivered

1. Ensure that you have a Linux operating system available.
2. Ensure that you have the Git version control system installed. For example, to install Git on Ubuntu:

   ```
   $ sudo apt install git
   ```

   For other OSes, see https://git-scm.com.
3. Clone the Reactors.IO [30, 32, 39] repository from GitHub:

   ```
   $ git clone \
     https://github.com/reactors-io/reactors.git
   ```

   The Reactors source code repository will appear in the subdirectory `reactors`.
4. (Optional) If installing the Git version control system was not successful, it is also possible to download the source code directly from GitHub. At the GitHub page of the project, look for a green button saying *Clone or download*, click on it, and press *Download ZIP*.

A.2.2 Hardware dependencies

You should have a quad-core (or better) x86 processor with hyperthreading. We advise using an Intel processor. We advise having at least 16GB of RAM memory.

A.2.3 Software dependencies

Ensure that you have the Oracle JDK 8 installed. For example, to install JDK 8 on Ubuntu:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Then follow the instructions shown on the screen. We advise using Ubuntu, since the installation process is very simple. However if you are using a different OS, see Oracle Downloads.

You should test if you have the correct JDK version by running the following:

```
$ java -version
java version "1.8.0_111"
Java(TM) SE Runtime Environment (build 1.8.0_111)
Java HotSpot(TM) 64-Bit Server VM
```

Above, the Java version must start with `1.8` (the minor versions are not important).

A.2.4 Data sets

There are no special datasets used in the evaluation. The workloads are generated automatically, as specified in the benchmark definitions (explained further below).

A.3 Installation

Enter the directory in which you cloned the Reactors source code repository (in part A.2.1). Inside that directory, run the `./sbt` script, which will start the `sbt` build tool:

```
$ ./sbt
```

The `sbt` build tool will lazily download all the dependencies for you. Make sure that you are connected to the Internet, and you are not behind a proxy or a VPN, and that you are not using an anti-virus software that blocks specific HTTP connections.

Once the `sbt` build tool is done loading (the first time, it might take a bit longer; subsequently it is fast), it will enter its own command shell. You will notice this by the fact that the prompt changes to `>`:

```
> _
```

At this point, you can compile the project. Please use the following command to compile the project:

```
> reactors-common-jvm/bench:compile
```

The `sbt` tool will download the appropriate version of the compiler, build the compiler interface, download the library dependencies, and then compile the project. Once this is successful, you will be able to run the benchmarks. You can test this by entering the following command, which will show level occupancy histograms of the cache trie data structure:

```
> reactors-common-jvm/bench:testOnly \
  io.reactors.common.concurrent.BirthdaySimulations
```

A.4 Experiment workflow

The experiment consists of three parts. In the first part, we test the hypothesis that most of the elements in the cache trie occupy some two adjacent levels. In the second part, we compare the memory footprint of the cache trie against the related data structures. In the third part, we compare the performance of different cache trie operations against the related data structures.

In each part, an experiment is run by invoking a specific command in the SBT shell. Each command runs a specific benchmark definition. Once the command completes, it will print information, such as the running time, in the terminal.

A.5 Evaluation and expected result

A.5.1 Level occupancy histograms

In the paper, we claim that the distribution of keys across levels of the cache trie data structure is such that most of the keys occupy some two adjacent levels. This property is formally proved in our technical report [31]. Our claim is that the *expected* number of keys in the two most populated levels is at least 87%. The property is important, since it allows using an auxiliary table that targets those two levels and speeds up searches in the cache trie.

To check this in practice, we prepared a command that iteratively creates cache tries with more and more keys, and prints the respective level occupancy histogram. The aim is to show that, regardless of the number of elements, there are some two levels that contain around 87% or more keys:

```
> reactors-common-jvm/bench:testOnly \
  io.reactors.common.concurrent.BirthdaySimulations
```

You should see a series of histograms, each for a different cache trie size. Here is an example histogram for a cache trie with 800000 elements, in which most of the keys are at levels 20 and 24 (note that level indices increment by 4).

```
:: size 800000 ::
  0:         0 (  0%)
  4:         0 (  0%)
  8:         0 (  0%)
 12:         0 (  0%)
 16:         8 (  0%)
 20:    370451 ( 46%) ******************
 24:    390164 ( 48%) *******************
 28:     38769 (  4%) *
 32:       608 (  0%)
```

The source code of this command can be found under the following path in the source code:

```
reactors/common/jvm/src/bench/scala/io/reactors/
common/concurrent/cache-trie-benches.scala
```

The test that prints out the histogram is defined at the end of the file in the class called `BirthdaySimulations`, in the test case called `"per level distribution"`.

A.5.2 Memory footprint

In Section 5, we showed that the memory footprint of cache tries is around $10\% - 25\%$ percent higher when the auxiliary table for speeding up searches gets added. The footprint of a cache trie is around $30\% - 50\%$ higher than that of a JDK concurrent hash map and concurrent skip list, depending on the number of keys stored.

To start the memory footprint measurements, please run the following command:

```
reactors-common-jvm/bench:testOnly \
  io.reactors.common.concurrent.CacheTrieFootprintBenches
```

In the final output, you will see the footprint of different data structures shown next to the `mean`. This an example output that we ran, which tests the memory footprint when 50000 keys are stored:

```
[info] - cache-trie.size.skiplist measurements:
[info]   - at size -> 50000: passed
[info]     (mean = 1798.92 kB, ...
[info] - cache-trie.size.ctrie measurements:
[info]   - at size -> 50000: passed
[info]     (mean = 2705.62 kB, ...
[info] - cache-trie.size.cachetrie measurements:
[info]   - at size -> 50000: passed
[info]     (mean = 2855.31 kB, ...
[info] - cache-trie.size.chm measurements:
[info]   - at size -> 50000: passed
[info]     (mean = 2121.54 kB, ...
```

The benchmark is set up to run for different data structure sizes, ranging from 100000 to 2000000 keys.

Please note that the memory footprint is usually a very stable, reproducible value, so this particular experiment is set up to do only 1 iteration of the warmup, and 4 measurement iterations.

A.5.3 Operation running time

Finally, in Section 5 we compared running times of cache tries and the related data structures. The most important observations were:

- Cache-trie lookups are $10\times - 33\times$ faster than skip lists.
- Cache-trie lookups are $2\times - 3\times$ faster than standard Ctries when there are $100k - 1M$ elements.
- Cache-trie lookups are $1.6\times - 2.0\times$ slower than concurrent hash map lookups.
- Cache-trie insertions have roughly the same performance as concurrent hash map insertions.
- Parallel cache-trie insertions scale better than concurrent hash map insertions.

These benchmarks take a bit longer than the memory footpring measurements. To start the running time measurements, please run the following command:

```
reactors-common-jvm/bench:testOnly \
  io.reactors.common.concurrent.CacheTrieBenches
```

This will run the following benchmarks:

- `cache-trie.apply`: Single-threaded lookup, number of elements $50k$ to $500k$.
- `cache-trie.insert`: Single-threaded insertion, number of elements $50k$ to $500k$.
- `cache-trie.par.lookup`: Parallel lookup, $100k$ elements, 1-8 threads.
- `cache-trie.par.insert`: Parallel insertion, $100k$ elements, 1-8 threads.

The output will be similar as before, with each benchmark annotated with its name, and consisting of performance reports for different data structures, and different number of keys. For example, the following part of the output refers to the parallel insertion benchmarks, and starts with the performance report for cache tries.

```
[info] Test group: cache-trie.par.insert
[info] - cache-trie.par.insert.cachetrie measurements:
[info]   - at pars -> 1: passed
[info]     (mean = 10.65 ms ...
```