Analysis of Concurrent Lock-Free Hash Tries with Constant-Time Operations

Aleksandar Prokopec
Oracle Labs
aleksandar.prokopec@gmail.com

Abstract

Ctrie is a scalable concurrent non-blocking dictionary data structure, with good cache locality, and non-blocking linearizable iterators [4]. However, operations on most existing concurrent hash tries run in $O(\log n)$ time. In this technical report, we extend the standard concurrent hash-tries [3] with an auxiliary data structure called a *cache*[1].

The cache is essentially an array that stores pointers to a specific level of the hash trie. We analyze the performance implications of adding a cache, and prove that the running time of the basic operations becomes $O(1)$.

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. Analysis of 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, 12 pages. https://doi.org/10.1145/3178487.3178498

1 Complexity Analysis

In this section, we show that the expected execution time of the cache-trie operations is $O(1)$. The proof consists of establishing the key depth distribution, and bounding the expected key and cache depths. From these bounds, we then conclude that the expected distance from the cache to the key is constant.

[1] A complete description of the new data structure is given in the corresponding PPoPP 2018 paper [2]

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

Definition 1.1. Depth d of a key is the number of pointer indirections required to reach an S-node by following pointers in the A-nodes minus 1, starting from the root. Level ℓ in a 16-way cache-trie is the number of hash code bits used to find the corresponding S-node, and is defined as $\ell = d \cdot 4$.

Definition 1.2. We say that a key with a specific hash code h *occupies* level ℓ or a depth $d = \ell/4$ in a cache-trie if it has a unique hash code ℓ-prefix in the cache-trie, but it does not have a unique $(\ell - 4)$-prefix.

Theorem 1.3. *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. Consider an arbitrary key x in the cache-trie. There are n other keys in the cache-trie, so the key x will occupy the level ℓ if some non-empty subset of k other keys has the same ℓ-prefix of the hash code, but not the same $(\ell+4)$-prefix, *and* the other $n - k$ keys have a different ℓ-prefix.

Next, note that, for any given set of keys S, the cache-trie has the same structure regardless of the order of insertion. Thus, we can behave as if the key x was the first key inserted into the cache-trie. The rest of the n keys are then inserted as n independent trials – each trial is an independent choice of a hash code, and can either cause a collision (have the same ℓ-prefix and a different $(\ell + 4)$-prefix compared to x), or not cause a collision (have a different ℓ-prefix compared to x). Two keys colliding at level ℓ have $\ell/4$ identical consecutive hash code substrings of length 4, followed by a different substring of length 4, so the corresponding probability is $16^{-\ell/4} \cdot 15/16$. The probability of a non-collision is $1 - 16^{-\ell/4}$. We are interested in those events in which there was at least one collision, so we count all combinations of k colliding keys, where $k \geq 1$.

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

From the identity:

$$\sum_{k=1}^{n} \binom{n}{k} P^k (1-P)^{n-k} Q^k = (1 - P + PQ)^n - (1-P)^n \tag{3}$$

we get:

$$p(\ell, n) = (1 - 16^{-\ell/4-1})^n - (1 - 16^{-\ell/4})^n \qquad (4)$$

By Definition 1.1, $d = \ell/4$, and the claim follows. □

The probability function $p(d, n)$ is a probability distribution over the depth d, as shown by the following corollary.

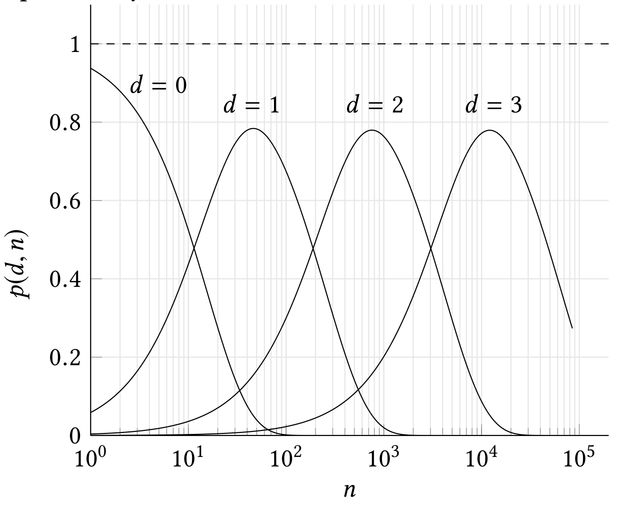
Corollary 1.4. *Let a cache-trie contain a fixed number of keys n. Then $p(d, n)$ is a discrete probability distribution over the depths d.*

Proof. From the definition of a discrete probability distribution – the sum of probabilities across all depths is 1:

$$\sum_{d=0}^{\infty} p(d,n) = (1 - \frac{1}{16})^n - 0 + (1 - \frac{1}{16^2})^n - (1 - \frac{1}{16})^n + \dots$$
$$= \lim_{d\to\infty} \Big[(1 - \frac{1}{16})^n + (1 - \frac{1}{16^2})^n - (1 - \frac{1}{16})^n +$$
$$+ \dots + (1 - \frac{1}{16^{d+1}})^n\Big] = \lim_{d\to\infty} (1 - 16^{-d-1})^n = 1$$

Furthermore, $p(d, n)$ is positive for a non-negative d. □

The following plot illustrates the probability $p(d, n)$ for different cache-trie sizes n, shown on the horizontal axis. Four different probability curves are shown for depths $d = 0$, $d = 1$, $d = 2$ and $d = 3$. The probability curve for $d = 0$ is initially close to 1, but then quickly drops to 0 before reaching 100 keys. The probability curves for larger depths start at 0, reach their maximum, and then descend back to 0. Note that the horizontal axis is logarithmic – the peeks are roughly exponentially distanced.



The previous plot suggests that for a given number of keys n contained in the cache-trie, most keys likely occupy a few adjacent depths. We will construct a function $\eta(d, n)$ that captures this notion, and then use it to construct another function $\mu(n)$ that estimates the proportion of keys contained at the most inhabited pair of depths.

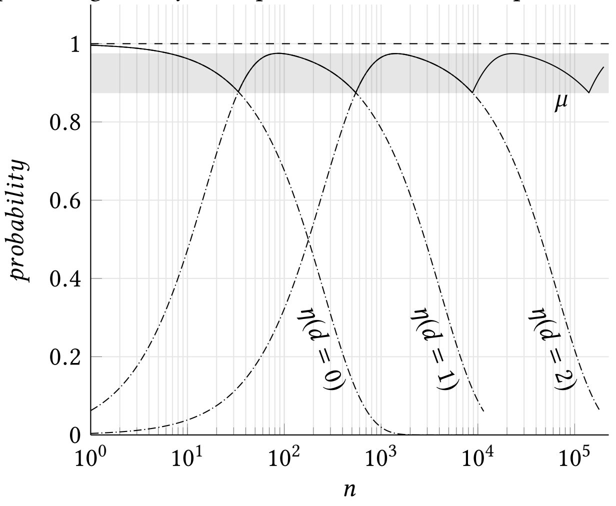
Definition 1.5. The η function returns the probability that a key occupies one of the consecutive depths d or $d + 1$, and is defined as follows:

$$\eta(d, n) = p(d, n) + p(d + 1, n) \qquad (5)$$

The μ function returns the probability that the key occupies the most inhabited pair of consecutive depths:

$$\mu(n) = \max_d \eta(d, n) \qquad (6)$$

Values of $\eta(d, n)$ and $\mu(n)$ are shown below in the following plot. It is now more apparent that, for any n, a large percentage of keys occupies two consecutive depths.



We now substantiate the intuition that the return values of μ are within a specific interval.

Theorem 1.6. *When the number of keys n tends to infinity, values of μ are inside the interval $\langle 0.8745, 0.9746\rangle$.*

Proof. We first prove the upper bound of the interval. The upper bound must be greater than or equal to all the maxima of $\eta(n, d)$.

$$0 = \frac{\partial\mu(n)}{\partial n} = \frac{\partial\eta(n,d)}{\partial n} = \frac{\partial}{\partial n}\Big[(1 - \frac{1}{16^{d+2}})^n - (1 - \frac{1}{16^d})^n\Big]$$
$$= (1 - \frac{1}{16^{d+2}})^n \ln(1 - \frac{1}{16^{d+2}}) - (1 - \frac{1}{16^d})^n \ln(1 - \frac{1}{16^d})$$

We get the following set of maxima, parametrized by d:

$$n_{max}(d) = \Big(\ln\frac{\ln(1 - 16^{-d})}{\ln(1 - 16^{-d-2})}\Big) \cdot \Big(\ln\frac{1 - 16^{-d-2}}{1 - 16^{-d}}\Big)^{-1} \qquad (7)$$

We can now compute the value of the maximum when n tends to infinity. Note that, for a specific d, $\eta(n, d)$ has a single maximum. Furthermore, $n_{max}(d)$ grows monotonically with d. Consequently, when n tends to infinity, d also tends to infinity. By puting $n_{max}(d)$ back into the expression for $\eta(n, d)$, and substituting $16^{-d} = t$, we get:

$$\mu_{upper} = \lim_{t\to 0}(1 - \frac{t}{256})^{n_{max}(t)} - (1-t)^{n_{max}(t)} \quad (8)$$

This limit can be easily simplified (we do not show the steps for brevity), and we get the following upper bound:

$$\mu_{upper} = 2^{-8/255} - 2^{-2048/255} \doteq 0.9746 \quad (9)$$

The upper bound is in itself not extremely useful, since we know that $\mu \leq 1$. The lower bound is more important, since it mandates the minimum proportion of keys that are close to the cache. As seen in the earlier plot for $\mu(n)$, the minimums occur when two $\eta(n,d)$ curves meet for two adjacent depths d. The number of keys n for which this happens is the solution to the following equation:

$$\begin{aligned}\eta(n, d+1) =& \eta(n,d)\\ (1-16^{-d-3})^n - (1-16^{-d-1})^n =& (1-16^{-d-2})^n - (1-16^{-d})^n\end{aligned}$$

This equation does not have an algebraic solution. Fortunately, we are only interested in how η behaves asymptotically for large d. Substituting $x = n \cdot 16^{-d}$, we get:

$$\begin{aligned}\lim_{d\to\infty} \eta(n, d+1) =& \lim_{d\to\infty} \eta(n,d)\\ \left[\sqrt[16^3]{\frac{1}{e}}\right]^x - \left[\sqrt[16^1]{\frac{1}{e}}\right]^x =& \left[\sqrt[16^2]{\frac{1}{e}}\right]^x - \left[\frac{1}{e}\right]^x\end{aligned}$$

This equation also does not have an algebraic solution, but we got rid of d, so we can solve it numerically. The solution $x_0 \doteq 34.315$ gives us the following lower bound:

$$\mu_{lower} = \lim_{d\to\infty} \eta(x_0 \cdot 16^d, d) \doteq 0.874553 \quad (10)$$

The solution given above is approximate, but it can be made arbitrarily precise, and is greater than 0.8745. □

Theorem 1.6 implies that there always exists a depth d such that a large percentage of the keys occupies the depth d or $d+1$. If the cache data structure targets this depth, then lookups and updates for those keys take $O(1)$ time. However, this does not yet prove the $O(1)$ bound on the expected running time – it is possible that some small percentage of keys are a non-costant number of levels deeper than the cache. In what follows, we prove that this is not the case – all the remaining keys are expected to occupy depths that are at most a constant number of levels away from the cache, regardless of the total number of keys n.

Lemma 1.7. *The following sum:*

$$S_1(t) = \sum_{j=1}^{t}(1-16^{-j})^{16^t} \quad (11)$$

converges and is less than $\frac{1}{e-1}$ when t tends to infinity.

Proof. To show this, we inspect the final term of the sum, and conclude that it converges to $\frac{1}{e}$ for large t. We then bound the sum with a geometric series.

$$\begin{aligned}\lim_{t\to\infty} S_1(t) =& \lim_{t\to\infty}\sum_{j=1}^{t}(1-16^{-j})^{16^t}\\ =& \lim_{t\to\infty} \ldots + (1-16^{-t+1})^{16^t} + (1-16^{-t})^{16^t}\\ =& \lim_{t\to\infty} \ldots + (1-16^{-t+1})^{16^{t-1}\cdot 16} + (1-16^{-t})^{16^t}\\ =& \ldots + \left(\frac{1}{e}\right)^{256} + \left(\frac{1}{e}\right)^{16} + \frac{1}{e}\\ <& \ldots + \left(\frac{1}{e}\right)^{4} + \left(\frac{1}{e}\right)^{3} + \left(\frac{1}{e}\right)^{2} + \frac{1}{e}\\ =& \frac{1}{e-1}\end{aligned}$$

□

Lemma 1.8. *The following sum:*

$$S_2(t) = \sum_{j=\log_{16} n}^{\infty}\left[1-(1-16^{-j})^n\right] \quad (12)$$

converges and is less than $\frac{1}{e-1}$ when n tends to infinity.

Proof. We rely on the limit when n tends to infinity to simplify the sum:

$$\begin{aligned}\lim_{n\to\infty} S_2(t) =& \lim_{n\to\infty}\sum_{k=1}^{\infty}\left[1-(1-16^{-k-\log_{16} n})^n\right]\\ =& \lim_{n\to\infty}\sum_{k=1}^{\infty}\left[1-(1-16^{-k}\cdot 16^{-\log_{16} n})^{n\cdot 16^k \cdot 16^{-k}}\right]\\ =& \sum_{k=1}^{\infty} 1 - \left(\frac{1}{e}\right)^{16^{-k}}\end{aligned}$$

We note that $e^{-k} \geq 1 - e^{-16^{-k}}$ when $k > 0$:

$$\begin{aligned}e^{-k} \geq& 1 - e^{-16^{-k}}\\ e^{\log_{16} x} \geq& 1 - e^{-x} \qquad k = -\log_{16} x\\ x^{(\ln 16)^{-1}} \geq& 1 - e^{-x}\\ x \geq& (1-e^{-x})^{\ln 16}\end{aligned}$$

The upper bound for S_2 follows:

$$\lim_{n\to\infty} S_2(t) \leq \sum_{k=1}^{\infty} e^{-k} = \frac{1}{e-1} \quad (13)$$

□

Theorem 1.9. *In a cache-trie that contains n keys, the expected depth of a key is bound by $\Theta(\log n)$. Moreover, the expected depth is exactly $E[d](n) = \log_{16} n + O(1)$.*

Proof. From the definition of the expected value of a random variable, for a specific choice of n:

$$\begin{aligned}
E[d](n) &= \sum_{d=0}^{\infty} d \cdot p(d,n) \\
&= \sum_{d=0}^{\infty} d \cdot \left[(1-16^{-d-1})^n - (1-16^{-d})^n\right] \\
&= 0 \cdot p(0,n) + (1-16^{-2})^n - (1-16^{-1})^n \\
&\quad + 2 \cdot (1-16^{-3})^n - 2 \cdot (1-16^{-2})^n \\
&\quad + 3 \cdot (1-16^{-4})^n - 3 \cdot (1-16^{-3})^n + \ldots \\
&= \lim_{d\to\infty} \left[d \cdot (1-16^{-d-1})^n + \sum_{j=1}^{d} (1-16^{-j})^n\right] \\
&= \lim_{d\to\infty} \sum_{j=1}^{d} (1-16^{-d-1})^n - (1-16^{-j})^n \\
&= \sum_{j=1}^{\infty} 1 - (1-16^{-j})^n
\end{aligned}$$

We separate the last sum into two parts at $j = \lceil \log_{16} n \rceil$:

$$E[d](n) = \sum_{j=1}^{\lceil \log_{16} n\rceil} \left[1-(1-16^{-j})^n\right] + \sum_{j=\lceil \log_{16} n\rceil+1}^{\infty} \left[1-(1-16^{-j})^n\right] \tag{14}$$

Consider the first sum, which consists of $\lceil \log_{16} n \rceil$ terms. We use Lemma 1.7 to compute the following lower bound for the expected depth, noting that the second sum in (14) is always positive and cannot affect the lower bound:

$$E[d](n) \geq \sum_{j=1}^{\lceil \log_{16} n\rceil} 1 - (1-16^{-j})^n \geq \log_{16} n - \frac{1}{e-1} \tag{15}$$

Coming back to (14), the first sum is upper bound by $\lceil \log_{16} n \rceil$. The second sum is less than $(e-1)^{-1}$ by Lemma 1.8, resulting in the following upper bound:

$$E[d](n) \leq \lceil \log_{16} n \rceil + \frac{1}{e-1} \tag{16}$$

From (15) and (16), $E[d](n) = \log_{16} n + O(1)$ follows. □

Theorem 1.10. *For any cache-trie, the expected distance between the cache depth and the key depth is* $O(1)$.

Proof. To emphasize the fact that we analyze the depths for a fixed number of keys n, we introduce a helper function $\eta_n(d)$, defined as follows:

$$\eta_n(d) = \eta(n,d) \tag{17}$$

We assume that the cache depth $E[d_{cache}]$ is chosen through an *unbiased sampling process* [1]. By definition, this implies that the expected sampled value $E[d_{sample}]$ of the most inhabited depth pair d corresponds to the true value of the most inhabited pair. The sampling strategy and the sample size influence the variance of the sample, but not its expected value.

$$E[d_{cache}](n) = E[d_{sample}](n) = \underset{d\in\mathbb{N}_0}{\operatorname{argmax}}\ \eta_n(d) \tag{18}$$

Therefore, the expected cache depth is equal to the true value of the most inhabited depth. It remains to show that the expected key depth $E[d](n)$ is $O(1)$ levels away from $\operatorname{argmax}_d \eta_n(d)$, i.e. the most inhabited depth pair. We will show that the expected cache depth is at most a small number of steps away from $\log_{16} n$. To do this, we consider how the function $\eta_n(d)$ behaves around the value $d = \log_{16}(n) - 2$ when n is large.

$$\begin{aligned}
\lim_{n\to\infty} \eta_n(\log_{16}(n) - 2) &= \lim_{n\to\infty} \left[(1-\frac{1}{n})^n - (1-\frac{256}{n})^{256^{-1}\cdot n\cdot 256}\right] \\
&= \frac{1}{e} - \left(\frac{1}{e}\right)^{256} \doteq \frac{1}{e}
\end{aligned}$$

Now, consider the expression for the first derivative of η_n:

$$\begin{aligned}
\frac{\partial \eta_n(d)}{\partial d} &= n \cdot (1-16^{-d-2})^{n-1} \cdot 16^{-d-2} \cdot \ln 16 \\
&\quad - n \cdot (1-16^{-d})^{n-1} \cdot 16^{-d} \cdot \ln 16
\end{aligned}$$

The first derivative of $\eta_n(d)$ is positive at $d = \log_{16}(n) - 2$:

$$\lim_{n\to\infty} \left.\frac{\partial \eta_n(d)}{\partial d}\right|_{\log_{16}(n)-2} = \left[\frac{1}{e} - 256 \cdot \left(\frac{1}{e}\right)^{256}\right] \cdot \ln 16 > 0$$

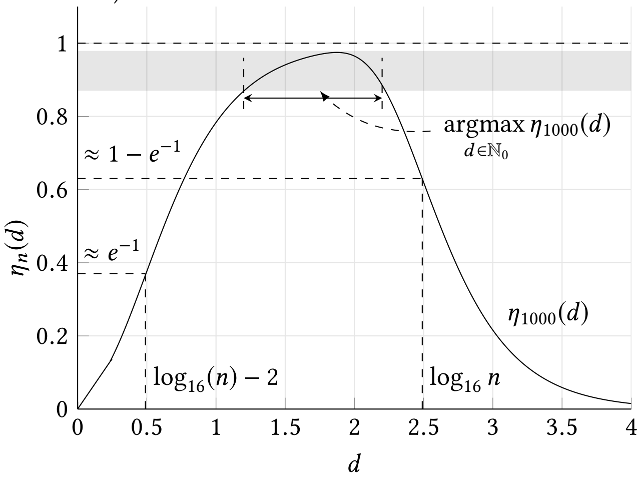
Similarly, we inspect $\eta_n(d)$ at $d = \log_{16} n$:

$$\begin{aligned}
\lim_{n\to\infty} \eta_n(\log_{16} n) &= \lim_{n\to\infty} \left[(1-\frac{1}{256n})^{256n\cdot 256^{-1}} - (1-\frac{1}{n})^n\right] \\
&= \left(\frac{1}{e}\right)^{256^{-1}} - \frac{1}{e} \doteq 1 - \frac{1}{e}
\end{aligned}$$

The first derivative is negative at $d = \log_{16} n$:

$$\lim_{n\to\infty} \left.\frac{\partial \eta_n(d)}{\partial d}\right|_{\log_{16} n} = \left[\frac{1}{256} \cdot \left(\frac{1}{e}\right)^{256^{-1}} - \frac{1}{e}\right] \cdot \ln 16 < 0$$

We conclude that the value d for which $\eta_n(d)$ achieves its maximum value must be within $\langle \log_{16}(n) - 2, \log_{16} n\rangle$, as illustrated by the following plot (for the purposes of illustration, we picked $n = 1000$, but the plot is similar for any choice of n).



By Theorem 1.6, $\mu(n) = \max_d \eta_n(d)$ must be within the interval $\langle 0.8745, 0.9746\rangle$, which is above the values $\frac{1}{e}$ and $1 - \frac{1}{e}$. The most inhabited depth pair must therefore also be within the same interval $\langle \log_{16}(n) - 2, \log_{16} n\rangle$. Consequently:

$$\underset{d\in\mathbb{N}_0}{\text{argmax}}\ \eta_n(d) = \log_{16}(n) + O(1) \tag{19}$$

By Theorem 1.9, the key depth $E[d](n)$ is $\log_{16}(n) + O(1)$. Therefore, $E[d_{cache}](n) = E[d](n) + O(1)$. ◻

Corollary 1.11. *Expected execution time of cache-trie lookup, insert and remove operations is $O(1)$.*

Proof. Direct consequence of the Theorem 1.10, and the fact that the cache-trie operations use the cache to find keys. ◻

Corollary 1.12 (Memory Footprint)**.** *The expected memory footprint of a cache-augmented cache-trie is $O(n)$.*

Proof. Theorem 1.9 implies that the expected path from the root the key is $O(\log n)$. Therefore, the expected memory footprint must be less than some complete 16-way tree of depth $O(\log n)$, which is $O(n)$. Furthermore, by Theorem 1.10, the cache-level is expected to be a constant number of levels away, so its expected memory footprint is $O(n)$ by the same argument. ◻

2 Correctness Proofs

We start by defining some preliminary notions, and then showing that cache-trie operations are safe. While proving safety, we develop sufficient foundation to easily show linearizability as well. After that, we show lock-freedom. The proofs assume the absence of the cache extension. We then prove that, by extending the cache-trie with the cache data structure, none of the previously proven properties change.

We start with some basic definitions.

Definition 2.1 (Data Types)**.** A *single node* (`SNode`) is a node that holds a single key and a transactional marker `txn`. For a node sn that holds the key k, the relation $key(sn, k)$ holds. An *array node* (`ANode`) is a node that contains a sequence of pointers to other nodes or `null` entries. A *narrow array node* contains 4 pointers or `null`s. A *wide array node* contains 16 pointers or `null`s. For an array node an, $length(an)$ is the number of pointers or `null`s that it contains, and $array(an, i)$ is the the entry at the index i. A *frozen node* (`FNode`) is a node that wraps an array node. For a frozen node fn, $unwrap(fn)$ is the node that it wraps. A *frozen single node* (`FSNode`) is a marker that denotes that a single node should not be modified. A *frozen vacant node* (`FVNode`) is a marker that denotes that an array node pointer is empty, and should no longer be modified. An *expansion node* (`ENode`) is a node that denotes that a narrow array node must be replaced with a wide one, and it holds a narrow array node, and a corresponding wide array node. For an expansion node en, $unwrap(en)$ is the narrow array node that it points to. Every node n can be assigned to a level ℓ, and this is denoted as n_ℓ. Every node n can be additionally assigned to a sequence of bits p, and this is denoted as $n_{\ell,p}$.

Definition 2.2 (Child Node)**.** For an array node an_ℓ at level ℓ and an index i, the pointer $child(an, i)$ is defined as follows:

$$child(an, i) = \begin{cases} null & array(an, i) \in \{\texttt{null}\} \\ cn & cn = array(an, i) \in \texttt{ANode} \\ unwrap(fn) & fn = array(an, i) \in \texttt{FNode} \\ unwrap(en) & en = array(an, i) \in \texttt{ENode} \\ null & array(an, i) \in \texttt{FVNode} \end{cases} \tag{20}$$

For convenience, we overload the $child$ relation for keys k. For an array node an_ℓ at level ℓ and a key k, the pointer $forKey(an_\ell, k)$ is defined as:

$$forKey(an_\ell, k) = array(an_\ell, (h{>>}\ell) \bmod length(an_\ell)) \tag{21}$$

where $h = hash(k)$.

For an array node an and a key k, the pointer $child(an, k)$ is defined as:

$$child(an, k) = \begin{cases} null & forKey(an, k) \in \{\texttt{null}\} \\ cn & cn = forKey(an, k) \in \texttt{ANode} \\ unwrap(fn) & fn = forKey(an, k) \in \texttt{FNode} \\ unwrap(en) & en = forKey(an, k) \in \texttt{ENode} \\ null & forKey(an, k) \in \texttt{FVNode} \end{cases} \tag{22}$$

Definition 2.3 (Cache-Trie)**.** A *cache-trie* is a pointer `root` to a wide array node. A *cache-trie* state $\mathbb{S}$ is the configuration of nodes reachable from the root by following the pointers of the nodes. A key k is contained in the state $\mathbb{S}$ if and only if a single node sn with the key k is reachable in the corresponding configuration. We define the relation $hasKey(an, k)$ for a node n and the key k as follows:

$$hasKey(an, k) \Leftrightarrow \begin{cases} sn = child(an, k) \in \texttt{SNode} \wedge key(sn) = k \\ cn = child(an, k) \in \texttt{ANode} \wedge hasKey(cn, k) \end{cases} \tag{23}$$

Definition 2.4 (Validity)**.** Let ϵ denote an empty sequence of bits, and $an_{x,y}$ denote an ANode. A cache-trie that respects the following invariants is called *valid*:
INV1 $\texttt{root} = an_{0,\epsilon} \in \texttt{ANode} \wedge length(\texttt{root}) = 16$
INV2 $child(an_{\ell,p}, i) \in \{an_{\ell+4,p\cdot i}, \texttt{null}\} \cup \texttt{SNode}$
INV3 $child(an_{\ell,p}, i) = sn \in \texttt{SNode} \Leftrightarrow hash(key(sn)) = p\cdot i\cdot s$

Definition 2.5 (Abstract Set)**.** An *abstract set* $\mathbb{A}$ is a mapping $\mathbb{A} : K \rightarrow \{\top, \bot\}$, where K is the set of all keys, and which is true for the keys that are present in the abstract set. *Abstract set operations* are:

- $lookup(\mathbb{A}, k) = \top \Leftrightarrow k \in \mathbb{A}$
- $insert(\mathbb{A}, k) = \mathbb{A}' : k \in \mathbb{A}' \wedge \forall k', k' \in \mathbb{A} \Rightarrow k' \in \mathbb{A}'$
- $remove(\mathbb{A}, k) = \mathbb{A}' : k' \notin \mathbb{A}' \wedge \forall k' \neq k, k' \in \mathbb{A} \Rightarrow k' \in \mathbb{A}'$

Definition 2.6 (Consistency)**.** A cache-trie state $\mathbb{S}$ is *consistent* with an abstract set $\mathbb{A}$ if and only if $\forall k, k \in \mathbb{A} \Leftrightarrow hasKey(\texttt{root}, k)$. The cache-trie lookup on the state $\mathbb{S}$ is consistent with an abstract set lookup if and only if for all keys k it returns the value $lookup(\mathbb{A}, k)$, where $\mathbb{A}$ is consistent with $\mathbb{S}$. The cache-trie insert and remove on the state $\mathbb{S}$ are consistent with an abstract set insert and remove, respectively, if an only if for all keys k they change the cache-trie to a new state $\mathbb{S}'$, such that $\mathbb{S}$ is consistent with an abstract set $\mathbb{A}$, and $\mathbb{S}'$ is consistent with an abstract set $\mathbb{A}'$, and $insert(\mathbb{A}, k) = \mathbb{A}'$, and $remove(\mathbb{A}, k) = \mathbb{A}'$, respectively.

Now that we have the basic definitions in place, we can state and prove the safety property.

Theorem 2.7 (Safety)**.** *At all times t, a cache-trie is valid and consistent with some abstract set $\mathbb{A}$. Cache-trie operations are always consistent with abstract set operations.*

Before proving this theorem, we state and prove several lemmas.

Definition 2.8 (Frozen Nodes)**.** A single node sn is *frozen* if its `txn` field is set to `FSNode`. The `FVNode` is always frozen. An array node an is *frozen* if all entries point to frozen nodes.

Lemma 2.9 (Single Transaction Change)**.** *A single node's* `txn` *field is initially set to* `NoTxn`*, and changes its value at most once.*

Proof. By inspecting the source code, we see that every `SNode` is created with `txn` set to `NoTxn`, and every `CAS` on `txn` has `NoTxn` as the expected value. The claim follows. □

Lemma 2.10 (End of Life)**.** *If an array node an is frozen at some time t_0, then $\forall t > t_0$ none of the entries of an change their value. If a single node sn gets removed from its parent at some time t_0, then its* `txn` *field was not set to* `NoTxn` *at time t_0.*

Proof. From the definition of a frozen array node, at t_0 all of its entries must be either a `FVNode`, `FNode` or an `SNode` with the `txn` field set to `FSNode`. Next, note that assignments to array node entries occur in `CAS` instructions. By inspecting these `CAS` instructions in the pseudocode, we can see that the expected value is never `FVNode` or `FNode`. Finally, note that when the expected value is an `SNode` sn (lines 17, 32 and 37 in Figure 3), the new value for the respective `CAS` is equal to sn's `txn` field. Assume that such a `CAS` succeeds. By Lemma 2.9, `txn` must be `FSNode`, since it can change at most once. That is a contradiction, since none of those `CAS` instructions has `FSNode` as the expected value. The argument applies inductively if an points to nested `ANode`s.

For the claim about the single node sn, note that only the `CAS` instructions in lines 17, 32, and 37 of Figure 3, and the `CAS` in line 20 of Figure 4, remove a single node from its parent. All of those instructions first check, that `txn` is not set to `NoTxn`. From Lemma 2.9, `txn` could not have changed back to `NoTxn` after the check, so the claim follows. □

Lemma 2.11 (Freezing)**.** *Let a call to the* `freeze` *subroutine return at some time t_0. Then, the array node passed to* `freeze` *is frozen at time t_0.*

Proof. Assume that the claim holds inductively for every nested `ANode`, and observe an entry at a particular index i. By analyzing the different cases, we see that the index i gets changed at the end of the loop iteration in `freeze` only if the respective entry is frozen. Hence, by the time that the loop in `freeze` completes, the respective array node is frozen. □

Lemma 2.12 (Unreachable Frozen Nodes)**.** *Let some* `CAS` *instruction make an array node an unreachable at time t_0. Then, that an was frozen at time t_0.*

Proof. The `CAS` instruction that makes an array node unreachable is in line 7 of the `completeExpansion` subroutine in Figure 4. From Lemma 2.11, we know that the expected value of the `CAS` instruction in line 7 is a frozen node. □

Lemma 2.13 (Reachable Nodes)**.** *If at some time t_0 a thread reads a node child from an array node an in line 4 of Figure 2 or in line 4 of Figure 3, then child is reachable at t_0.*

Proof. Assume the opposite – that the node $child$ is not reachable at t_0. Then, by Lemma 2.12, an was frozen at t_0. But that is a contradiction, since at t_0 the thread read an array node, not an `FNode`. □

Lemma 2.14 (Presence)**.** *If a thread reads a single node sn at some time t_0 from an array node an, in line 4 of Figure 2 or in line 4 of Figure 3. then the relation $hasKey(\texttt{root}, key(sn))$ holds at the time t_0.*

Proof. From Lemma 2.13, we know that the single node *sn* was reachable at time t_0. The claim follows from the definition of the *hasKey* relation. □

Definition 2.15 (Path). A *path* $\pi(h)$ for some hash-code h is a sequence of nodes from the root to a leaf, such that:

- The first node in the path is $an_{0,\epsilon}$ = `root`.
- $\forall an_{\ell,p} \in \pi(h)$, if $h = p \cdot i \cdot s$ and $child(an, i) = cn$, then the next element of the path is *cn*.
- $\forall an_{\ell,p} \in \pi(h)$, if $h = p \cdot i \cdot s$ and $child(an, i) = null$, then *an* is the last element of the path.

Lemma 2.16 (Path Form). *Let the cache-trie be in a valid state* $\mathbb{S}$. *The path* $\pi(h)$ *for some hash-code* $h = i_0 \cdot i_1 \cdot \ldots \cdot i_n \cdot s$ *is finite and has the form* $an_{0,\epsilon} an_{4,i_0} \ldots an_{4n,i_0 \cdot \ldots \cdot i_n} x$, *where* x *is either empty or* $sn \in$ `SNode`.

Proof. Directly from the definition and the invariants of the valid cache-trie state. □

Lemma 2.17 (Absence I). *Let* $hash(k) = p \cdot i \cdot s$ *for some key* k. *Assume that at some time* t_0 *a thread is searching for a key k and it reads* `null` *from an array node* $an_{\ell,p}$ *in line 4 of Figure 2 or in line 4 of Figure 3. Then* $hasKey($`root`$, k)$ *does not hold at time* t_0.

Proof. By the inductive hypothesis, cache-trie was valid and consistent at the time t_0. By Lemma 2.13, $an_{\ell,p}$ is reachable at time t_0. Now, consider the path $an_{0,\epsilon} \to an_{4,i_0} \to \ldots \to an_{\ell,p}$ consisting of nodes that the thread traversed while searching for k, where $hash(k) = i_0 \cdot i_1 \cdot \ldots \cdot i_n \cdot s = p \cdot s$. By Lemma 2.13, none of those nodes could have been removed from the cache-trie between the time they were read and the time t_0. Therefore, by Lemma 2.16, if the cache-trie state contains the key k, then the path to the respective single node *sn* should have the prefix $an_{0,\epsilon} \to \ldots \to an_{\ell,p}$. To show this, assume by contradiction that there is a key *sn* that does not have this path prefix. This assumption violates the inductive hypothesis that the cache-trie is valid and consistent. □

Lemma 2.18 (Absence II). *Assume that at some time* t_0 *a thread is searching for a key k and it reads single node* `sn` *such that* $k \neq key(sn)$. *Then, the relation* $hasKey($`root`$, k)$ *does not hold at* t_0.

Proof. Similar to Lemma 2.17. □

Lemma 2.19 (Fastening). *Assume that the cache-trie is valid and consistent with some abstract set.*

1. *Assume that the* `CAS` *instruction in line 32 (Figure 3) succeeds with the expected value* `old` *at some time* t_1 *after* `old` *was read from the entry* `pos` *of the array node* `cur` *in line 4 at time* $t_0 < t_1$. *Then,* $\forall t \in \langle t_0, t_1 \rangle$, *the relation* $hasKey($`root`, `k`$)$ *does not hold.*
Otherwise, if the `CAS` *instruction in line 32 does not succeed, then there is a duration* $\delta > 0$, *such that during* $\langle t_0, t_0 + \delta \rangle$ *the relation* $hasKey($`root`, `k`$)$ *does not hold, but at* $t_0 + \delta$ *it holds. At* $t_0 + \delta$ *either the* `CAS` *in line 37 (Figure 3) or the* `CAS` *in line 20 (Figure 4) succeeds.*
2. *Assume that the* `CAS` *instruction in line 17 (Figure 3) succeeds with the expected value* `old` *at some time* t_1 *after* `old` *was read from the entry* `pos` *of the array node* `cur` *in line 4 at time* $t_0 < t_1$. *Then,* $\forall t \in \langle t_0, t_1 \rangle$, *the relation* $hasKey($`root`, `k`$)$ *holds.*
Otherwise, if the `CAS` *instruction in line 17 does not succeed, then there is a duration* $\delta > 0$, *such that during* $\langle t_0, t_0 + \delta \rangle$ *the relation* $hasKey($`root`, `k`$)$ *holds, and continues to hold after* $t_0 + \delta$, *but* `old` *is not reachable at* $t_0 + \delta$. *At* $t_0 + \delta$ *either the* `CAS` *in line 37 (Figure 3) or the* `CAS` *in line 20 (Figure 4) succeeds.*
3. *Assume that the* `CAS` *instruction in line 7 (Figure 3) succeeds with the expected value* `null` *at some time* t_1 *after* `old` *was read from the entry* `pos` *of the array node* `cur` *in line 4 at time* $t_0 < t_1$. *Then, there exists some duration* $\delta > 0$ *such that* $\forall t \in \langle t_1 - \delta, t_1 \rangle$, *the relation* $hasKey($`root`, `k`$)$ *does not hold.*

Proof. We proceed casewise:

1. By Lemma 2.13, we know that `cur` was reachable at time t_0. First assume that the `CAS` in line 32 was successful. Then, the `txn` field of `old` must have been set to the same value that `CAS` succeeded with. By Lemma 2.9, we know that this was the only value that was ever written to `txn`. The conclusion is that `old` was at the position `cur[pos]` during the interval $\langle t_0, t_1 \rangle$. Furthermore, by Lemma 2.16, `old` is the only single node in the cache-trie that could hold the key `k`. Since we check that `old.key` is not equal `k`, the conclusion is that $hasKey($`root`, `k`$)$ does not hold during $\langle t_0, t_1 \rangle$.
Now assume that the `CAS` in line 32 was not successful. This means that some other thread helped by committing the `txn` value. In other words, there was at least one other `CAS` instruction that changed `cur[pos]` between the `CAS` in line 31 and the `CAS` in line 32. By Lemma 2.10, the first such `CAS` instruction must have been in line 37 (Figure 3) or in line 20 (Figure 4).
2. The reasoning is similar as in the previous case – the `old` node must have been present during $\langle t_0, t_1 \rangle$. This time, since we checked that `old.key` is equal to `k`, $hasKey($`root`, `k`$)$ holds during $\langle t_0, t_1 \rangle$. Again, if the `CAS` was not successful, then it means that some other thread helped in commiting the transaction during $\langle t_0, t_1 \rangle$.
3. Since the `CAS` in line 7 succeeded, the entry `cur[pos]` was `null` during $\langle t_1 - \delta, t_1 \rangle$. Note that the node `cur` was reachable during $\langle t_0, t_1 \rangle$, since it was not frozen (Lemmas 2.10 and 2.12). Therefore, by Lemma 2.16, the single node with a key `k` can only appear at `cur[pos]`, and the claim follows.

□

Lemma 2.20 (Consistency Changes). *Assume that the cache-trie is valid and consistent with some abstract set $\mathbb{A}$. Then,* `CAS` *instructions from Lemma 2.19 induce a change into a valid state that is consistent with the abstract set semantics.*

Proof. Follows directly from Lemma 2.19 and the definition of consistency. □

Lemma 2.21 (Housekeeping). *Let the cache-trie be valid and consistent with some abstract set $\mathbb{A}$. Assume that one of the* `CAS` *instructions in lines 16, 23, and 31 of Figure 3, or in lines 5, 7, 14, 18, and 18 of Figure 4, succeed. Then, the cache-trie remains valid and consistent with $\mathbb{A}$ after those instructions succeed.*

Proof. From the definition of the *hasKey* relation, it is straightforward to see that node of these `CAS` instructions are state-changing. □

Corollary 2.22 (Invariant Preservation). *Cache-trie invariants are always preserved.*

Proof. From Lemmas 2.20 and 2.21. □

Proof of Theorem 2.7. From Lemmas 2.14, 2.17, 2.18, 2.20 and 2.21, and Corollary 2.22. □

Theorem 2.23 (Linearizability). *The operations* `lookup` *and* `insert` *are linearizable.*

Proof. An operation is linearizable if, from the perspective of all the threads in the system, there is a single point in time during the execution of that operation at which the cache-trie changes consistency.

We already identified the `CAS` instructions that change the consistency in Lemma 2.20. We also identified the `CAS` instructions that do not change the consistency in Lemma 2.21. It remains to show that during an execution of any operation, there is at most one successful state-changing `CAS` instruction that is consistent with the abstract set semantics of that operation. We consider all successful state-changing `CAS` instructions in `insert`, and show that they either induce a state-change of a concurrent operation, or they are the last successful state-changing `CAS` instructions in the current operation.

- It is easy to see that following a successful `CAS` in lines 7, 17, and 32 (Figure 3), the respective `insert` invocation ends in a finite number of steps without running any other `CAS` instruction.
- Successful `CAS` instructions in lines in line 37 (Figure 3) and 20 (Figure 4) induce a state-change in a concurrent `insert` operation, since they must occur after the successful `CAS` in lines 16 or 31, and before the failed `CAS` in lines 17 or 32, respectively.

Finally, note that no code path in `insert` ends the operation without at least one such successful `CAS` operation. □

Theorem 2.24 (Lock-Freedom). *The operations* `lookup` *and* `insert` *are lock-free.*

Lemma 2.25. *In each operation, there is a finite number of steps between two* `CAS` *instructions, a* `CAS` *instruction and the entry point or a return point.*

Proof. The only `while` loop in the pseudocode is in the `freeze` subroutine, which has a finite upper bound, and executes a `CAS` whenever it decreases its counter.

Assume, therefore, that there exists an infinite number of execution steps caused by recursion in `insert`. The recursion never decreases the cache-trie level, and retains the same level if and only if a `CAS` fails. We know from Lemma 2.16 that each path in the cache-trie is finite, so the level cannot grow indefinitely. Thus, there cannot be an infinite number of execution steps without a `CAS`. □

Lemma 2.26. *If any* `CAS` *instruction C_0 fails, then there was a concurrent successful* `CAS` *instruction C_1 on the same memory location that executed between the time t_r when the expected value for C_0 was read, and the time $t_0 > t_r$ when C_0 failed.*

Proof. This can be easily proven by examining all the `CAS` instructions and their expected values. □

Corollary 2.27. *From Lemmas 2.25 and 2.26, it follows that there is a finite number of execution steps between any two state changes.*

A change in the cache-trie state does not imply a consistency change. For example, when some operation expands a narrow node into a wide node, the same set of keys remain in the cache-trie. We therefore need to bound the number of subsequent state changes that do not cause a consistency change – we will show that, eventually, a state change must also change the abstract set that the cache-trie is consistent with.

Definition 2.28 (Transaction Potential). Given a cache-trie in some state $\mathbb{S}$, the *transaction potential* $P(\mathbb{S})$ is the number of single nodes whose `txn` field is set to `NoTxn`.

Definition 2.29 (Expansion Potential). Given a cache-trie in some state $\mathbb{S}$, the *expansion potential* $E(\mathbb{S})$ is the number of narrow array nodes.

Lemma 2.30. *Non-consistency-changing* `CAS` *instructions always either decrease the expansion potential or decrease the transaction potential.*

Proof. Straightforward, by enumerating the `CAS`es. □

Lemma 2.31. *Consider the consistency-changing* `CAS` *instructions, and all* `CAS` *instructions that cause a decrease in $P(\mathbb{S}) + E(\mathbb{S})$. There is a finite number of steps between two such instructions.*

Proof. Assume that consistency-changing `CAS` instructions never occur. By inspecting the remaining state-changing `CAS` instructions, we see that each of them either expands a node (i.e. decreases $E(\mathbb{S})$) or it announces a transaction (i.e. decreases $P(\mathbb{S})$). Therefore, after a finite number of steps, $P(\mathbb{S}) + E(\mathbb{S}) = 0$. This is a contradiction, since now a consistency-changing CAS must occur. □

Proof of Theorem 2.24. From Corollary 2.27 and Lemmas 2.30 and 2.31 – since in any cache-trie state, there is a finite number of unfinished transactions and narrow nodes, there is a finite number of steps between consistency changes. By Theorem 2.23, each consistency change corresponds to a completed operation. □

Definition 2.32 (Cache)**.** A *cache node* is a node that holds a pointer to a parent cache, and an integer array for tracking cache misses. A *cache array* is an array of length $2^\ell + 1$, that contains a cache node at the index 0, and the remaining entriese are either `SNodes` or `ANodes`. We say that such a cache array is at level ℓ. A *cache* is a pointer that is either `null` or points to a cache array.

Lemma 2.33 (Cachee Level)**.** *A cache array at level ℓ contains pointers to nodes that are either frozen or have* `txn` *different than* `NoTxn`*, or are reachable and at level ℓ of the cache-trie (except at the index* 0*).*

Proof. Consider the only write to the cache in line 13 of Figure 7. The write is preceded by a check that the cache level corresponds the cachee level. Any `SNode` or `ANode` in the cache-trie that is made unreachable must be frozen by Lemma 2.12, and frozen nodes are never made reachable once they become unreachable. Therefore, a reachable node remains at level ℓ of the cache. □

Theorem 2.34 (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.*

Proof. We start by considering consistency. From Lemma 2.33, we know that a non-frozen node or a single node whose `txn` is different than `NoTxn` is reachable in the cache-trie.

Assume that the fast lookup or fast insert calls the normal lookup or insert, e.g. in line 39. Since the respective array node is reachable, safety and linearizability follow immediately by the same reasoning as Theorems 2.7 and 2.23. Since the total number of execution steps is smaller compared to an execution in which the search started at the root, lock-freedom follows as well, by Theorem 2.24.

Assume that the fast lookup returns a value in line 31, or `null` in line 32 of Figure 6. This is preceded by the read of the `txn` field at some time t_0, in line 29. Since `txn` is `NoTxn` at t_0, the corresponding node is reachable, and by Lemma 2.16, the relation *hasKey* must hold. Therefore, fast lookup is safe. The read of the `txn` field is the linearization point, so fast lookup is linearizable. Since it takes a finite number of steps to reach that point, fast lookup is also lock-free.

Finally, consider the case in which the node read from the cache is frozen or has `txn` different than `NoTxn`. In this case, a fast lookup or a fast insert both fall back to a normal, so all three properties hold, by Theorems 2.7, 2.23 and 2.24. □

References

[1] W.G. Cochran. 1977. *Sampling Techniques.* Wiley.

[2] Aleksandar Prokopec. 2018. Cache-Tries: Concurrent Lock-Free Hash Tries with Constant Time Operations. (2018), 11.

[3] 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

[4] 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

```
class SNode                  class FNode
  val hash: Int                val frozen: Any
  val key: KeyType
  val value: ValueType       class ENode
  var txn: Any                 val parent: ANode
                               val parentpos: Int
type ANode = Array<Any>        val narrow: ANode
                               val hash: Int
val NoTxn                      val level: Int
                               var wide: ANode
val FSNode
                             class CacheTrie
val FVNode                     val root = new ANode(16)
```

Figure 1. Basic Cache-Trie Data Types

```
def lookup(key: KeyType, hash: Int, level: Int,
  cur: ANode): ValueType =
  val pos = (hash >>> level)⊙(cur.length - 1)
  val old = READ(cur[pos])
  if (old == null ∨ old == FVNode)
    return null
  else if (old ∈ ANode)
    return lookup(key, hash, level + 4, old)
  else if (old ∈ SNode)
    if (old.key == key) return old.value
    else return null
  else if (old ∈ ENode)
    val an = old.narrow
    return lookup(key, has, level + 4, an)
  else if (old ∈ FNode)
    return lookup(key, hash, level + 4, old.frozen)

def lookup(key: KeyType): ValueType =
  lookup(key, hash(key), 0, root)
```

Figure 2. Lookup Operation

A Operation Pseudocode

This section contains the pseudocodes of the cache-trie operations. For a full discussion, we refer the readers to the corresponding paper [2].

```
def insert(k: KeyType, v: ValueType, h: Int,
  lev: Int, cur: ANode, prev: ANode): Boolean =
  val pos = (h >>> lev)⊙(cur.length - 1)
  val old = READ(cur[pos])
  if (old == null)
    val sn = new SNode(h, k, v, NoTxn)
    if (CAS(cur[pos], old, sn)) return true
    else return insert(k, v, h, lev, cur, prev)
  else if (old ∈ ANode)
    return insert(k, v, h, lev + 4, old, cur)
  else if (old ∈ SNode)
    val txn = READ(old.txn)
    if (txn == NoTxn)
      if (old.key == key)
        val sn = new SNode(h, k, v, NoTxn)
        if (CAS(old.txn, NoTxn, sn))
          CAS(cur[pos], old, sn)
          return true
        else return insert(k, v, h, lev, cur, prev)
      else if (cur.length == 4)
        val ppos = (h >>> (lev - 4))⊙(prev.length - 1)
        val en = new ENode(prev, ppos, cur, h, lev)
        if (CAS(prev[ppos], cur, en))
          completeExpansion(en)
          val wide = READ(en.wide)
          return insert(k, v, h, lev, wide, prev)
        else return insert(k, v, h, lev, cur, prev)
      else
        val sn = new SNode(h, k, v, NoTxn)
        val an = createANode(old, sn, lev + 4)
        if (CAS(old.txn, NoTxn, an))
          CAS(cur[pos], old, an)
          return true
        else return insert(k, v, h, lev, cur, prev)
    else if (txn == FSNode) return false
    else
      CAS(cur[pos], old, txn)
      return insert(k, v, h, lev, cur, prev)
  else if (old ∈ ENode) completeExpansion(old)
  return false

def insert(k: KeyType, v: ValueType) =
  if (!insert(k, v, hash(k), 0, root, null))
    insert(k, v)
```

Figure 3. Insert Operation

```
def completeExpansion(en: ENode) =
  freeze(en.narrow)
  var wide = new Array<Any>(16)
  copy(en.narrow, wide, en.level)
  if (!CAS(en.wide, null, wide))
    wide = READ(en.wide)
  CAS(en.parent[en.parentpos], en, wide)

def freeze(cur: ANode) =
  var i = 0
  while (i < cur.length)
    val node = READ(cur[i])
    if (node == null)
      if (!CAS(cur[i], node, FVNode)) i -= 1
    else if (node ∈ SNode)
      val txn = READ(node.txn)
      if (txn == NoTxn)
        if (!CAS(node.txn, NoTxn, FSNode)) i -= 1
      else if (txn ≠ FSNode)
        CAS(cur[i], node, txn)
        i -= 1
    else if (node ∈ ANode)
      val fn = new FNode(node)
      CAS(cur[i], node, fn)
      i -= 1
    else if (node ∈ FNode)
      freeze(node.frozen)
    else if (node ∈ ENode)
      completeExpansion(node)
      i -= 1
    i += 1
```

Figure 4. Freezing and Expansion

```
type Cache = Array<Any>

class CacheNode
  val parent: Array<Any>
  val misses: Array<Int>

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

def createCache(level: Int, parent: Cache): Cache =
  val cache = new Array(1 + (1 << level))
  val misses = new Array(THROUGHPUT_FACTOR * #CPU)
  cache[0] = new CacheNode(null, 8, misses)
  return cache
```

Figure 5. Cache Data Types and Helper Functions

```
def lookup(k: KeyType, hash: Int, lev: Int,
  cur: ANode, lastCachee: Any, cacheLevel: Int) =
  if (lev == cacheLevel)
    inhabit(cache, cur, hash, lev)
  val pos = position(cur, hash, lev)
  ...
  else if (old ∈ SNode)
    if (lev ∉[cacheLevel, cacheLevel + 4])
      recordCacheMiss()
    if (lev + 4 == cacheLevel)
      inhabit(cache, old, hash, lev + 4)
    if (old.key == key)
  ...
      return lookup(key, hash, level + 4, old.frozen)

def fastLookup(k: KeyType): ValueType =
  val h = hash(k)
  var cache = READ(cacheHead)
  if (cache == null)
    return lookup(k, h, 0, root, null, -1)
  val topLevel = countTrailingZeros(cache.length - 1)
  while (cache ≠ null)
    val pos = 1 + (h⊙(cache.length - 2))
    val cachee = READ(cache[pos])
    val level = countTrailingZeros(cache.length - 1)
    if (cachee ∈ SNode)
      val txn = READ(old.txn)
      if (txn == NoTxn)
        if (cachee.key == k) return cachee.value
        else return null
    else if (cachee ∈ ANode)
      val cpos = (h >>> level)⊙(cachee.length - 1)
      val old = READ(cachee[cpos])
      if (old == FVNode ∨ old ∈ FNode) continue
      if (old ∈ SNode)
        if (READ(old.txn) == FSNode) continue
      return lookup(k, h, level, cachee, level)
    cache = cache[0].parent
  return lookup(k, h, 0, root, null, topLevel)
```

Figure 6. Modified Lookup and the Fast Lookup Operation

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

Figure 7. Inhabiting the Cache

```
def recordCacheMiss() =
  val cache = READ(cacheHead)
  if (cache ≠ null)
    val cn = cache[0]
    val counterId = THREAD_ID % cn.misses.length
    val count = READ(cn.misses[counterId])
    if (count > MAX_MISSES)
      WRITE(cn.misses[counterId], 0)
      sampleAndAdjustCache(cache)
    else WRITE(cn.misses[counterId], count + 1)

def sampleAndAdjustCache(cache: Array<Any>) =
  val histogram = sampleSNodesLevels()
  val best = findMostPopulatedLevel(histogram)
  val prev = countTrailingZeros(cache.length - 1)
  if (histogram[best] > histogram[prev] * 1.5)
    adjustCacheLevel(best)
```

Figure 8. Recording Cache Misses and Sampling