CAFÉ: Scalable Task Pools with Adjustable Fairness and Contention

Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov and Dmitri Perelman

CCIT Report #790
May 2011

CCIT Report #790 May 2011

CAFÉ: Scalable Task Pools with Adjustable Fairness and Contention

Dmitry Basin[*] Rui Fan[†] Idit Keidar[*] Ofer Kiselov[*] Dmitri Perelman[*]

Abstract

Task pools have many important applications in distributed and parallel computing. Pools are typically implemented using concurrent queues, which limits their scalability. We introduce *CAFÉ, Contention and Fairness Explorer*, a scalable and wait-free task pool which allows users to control the trade-off between fairness and contention. The main idea behind CAFÉ is to maintain a list of *TreeContainers*, a novel tree-based data structure providing efficient task inserts and retrievals. TreeContainers don't guarantee FIFO ordering on task retrievals. But by varying the size of the trees, CAFÉ can provide any type of pool, from ones using large trees with low contention but less fairness, to ones using small trees with higher contention but also greater fairness.

We demonstrate the scalability of TreeContainer by proving an $O(\log^2 N)$ bound on the step complexity of insert operations when there are N inserts, as compared to an average of $\Omega(N)$ steps in a queue based implementation. We further prove that get operations are wait-free. Evaluations of CAFÉ show that it outperforms the Java SDK implementation of the Michael-Scott queue by a factor of 30, and is over three times faster than other state-of-the-art non-FIFO task pools.

[*]Department of Electrical Engineering, Technion, Haifa, Israel {sdimbsn@tx,idish@ee,sopeng@t2,dima39@tx}.technion.ac.il
[†]School of Computer Engineering, Nanyang Technological University, fanrui@ntu.edu.sg

1 Introduction

A *task pool* is a data structure consisting of an unordered collection of objects, a *put* operation to add an object to the collection, and a *get* operation to remove an object[1]. Pools have a number of important applications in multiprocessor computing, such as maintaining the set of pending tasks in a parallel computation. A key challenge in such an application is to ensure the pool does not become a bottleneck when it is concurrently accessed by a large number of threads. Another challenge is to ensure fairness — although strict FIFO ordering is not necessary, we nevertheless want to avoid starvation and limit the number of *overtakings*[2].

In this paper, we present CAFÉ (Contention And Fairness Explorer), an efficient randomized wait-free[3] task pool algorithm. CAFÉ maintains a list of scalable bounded pools called *TreeContainers*. When one TreeContainer becomes full, a new TreeContainer is appended to the end of the list. Retrievals follow the FIFO order of the TreeContainers, but each TreeContainer can return its tasks in any order. This way, the tree size is a system parameter controlling the trade-off between fairness and contention. Using smaller trees, the system provides better fairness but also has more contention.

A TreeContainer stores jobs in a complete binary tree, in which every node can store one task. Each node keeps presence bits indicating whether its child subtrees contain tasks. This allows get operations to find tasks by walking down the tree from the root, following a trail of presence bits. At the same time, the bits do not change frequently, even when there are a large number of concurrent puts and gets, so they do not cause much contention. We show that TreeContainers are dense: a tree with height h contains at least $2^{(1-\epsilon)h}$ tasks with high probability, for any $\epsilon > 0$. We also show that TreeContainers perform well under contention. When there are N concurrent put operations and an arbitrary number of gets, each put finishes in $O(\log^2 N)$ steps, whp.

CAFÉ combines TreeContainers in a FIFO linked list, to provide the following properties. 1) The number of overtaken tasks in CAFÉ is bounded by the size of a tree. 2) In most workloads, producers and consumers operate on different TreeContainers, which decreases contention and improves performance. 3) Puts are wait-free with probability 1, and gets are deterministically wait-free.

Our algorithm offers some significant advantages over other approaches for task pools. The most common approach to implement pools is using FIFO queues (e.g., Java ThreadPoolExecutor). However, non-blocking queue-based algorithms suffer $\Omega(N)$ contention at the head and tail, while our algorithm has $O(\log^2 N)$ contention for puts, whp. Other queue-based algorithms are blocking, and require puts and gets to wait for each other. In contrast, all operations in our algorithm are wait-free. The recent ED pools in [1] also use trees, but in a different way. Unlike our algorithm, [1] does not provide any upper bounds on step complexity, nor on the number of times a task can be overtaken.

We have implemented CAFÉ in Java, and tested its performance on a 32-core machine[4]. Our results show that CAFÉ is over 30 times faster than a pool based on Java's implementation of the Michael-Scott queue, and over three times faster than a pool using Java's state-of-the-art blocking queue (even though CAFÉ does not block). Also, CAFÉ is over three times faster than ED pools, while providing stronger fairness guarantees.

The remainder of the paper is organized as follows. In Section 2, we describe related work. We present CAFÉ in Section 3, and analyze its theoretical properties in Section 4. We discuss our experimental results in Section 5. Finally, we conclude in Section 6.

[1]We sometimes refer to task pools as producer-consumer pools; producers do puts, and consumers do gets.

[2]One task overtakes another task if it is inserted after the other task, but retrieved before it.

[3]A randomized algorithm is *wait-free* if each thread executing an operation performs a finite number of steps with probability 1.

[4]The code is publicly available at `http://code.google.com/p/cafe-pool/`.

2 Related Work

A common approach to implementing concurrent task pools is to use FIFO queues for task management. However, due to their strong ordering guarantees, such implementations are not scalable, suffering from $\Omega(N)$ contention in the worst case. CAFÉ makes the observation that strict FIFO ordering is not necessary for a task pool, and thereby achieves a much more scalable algorithm.

Another approach for reducing contention is using *elimination*, as proposed by Moir et al. [7]. Here, producers and consumers can "eliminate" each other at predefined rendezvous points. This approach best suits workloads in which there are more consumers than producers. Elimination is less useful if the queue remains non-empty most of the time, or when concurrency is low. In contrast, CAFÉ performs well under both high and low concurrency, and regardless of the ratio between producers and consumers.

Afek *et al.* [1] also propose a task pool foregoing FIFO ordering for scalability. Their Elimination Diffraction (ED) pools yield significantly better results than FIFO implementations. ED pools use a fixed number of queues along with elimination for reducing contention. However, as we show in Section 5.2, ED pools do not scale well on multi-chip architectures. In addition, unlike CAFÉ , ED pools are not wait-free, and offer no fairness guarantees between puts and gets.

The idea of using concurrent tree-based data structures for reducing contention has appeared in previous works not related to task pools [3, 2]. Unlike these works, we prove formal bounds on the worst case step complexity of our TreeContainer algorithm.

3 CAFÉ: A Task Pool with Adjustable Fairness

In this section, we describe CAFÉ, a wait-free, scalable task pool algorithm, whose fairness can be adjusted arbitrarily by the user. The main idea behind CAFÉ is to keep a linked list of scalable task pools called *TreeContainers*, each with bounded size. The algorithm for a single TreeContainer is given in Section 3.1. Tasks are stored at tree nodes, which can be occupied at most once. When a tree becomes full, a new tree is added to the list. The algorithm for combining TreeContainers in a FIFO list is described in Section 3.2.

3.1 TreeContainer

A TreeContainer consists of a bounded complete binary tree, in which each node can store one task. A node with a task is *occupied*, and otherwise it is *free*. Each node can be occupied at most once, as indicated by an *isDirty* flag. In addition, the node keeps a *presence bit* for each child subtree; the bit is zero when all the nodes in the respective subtree are free. Presence bits allow get operations to find a task in the tree by walking down from the root following a trail of non-zero bits. Since presence bits summarize the occupancy of an entire subtree, they change infrequently even under highly concurrent workloads, which allows our algorithm to achieve low step complexity.

TreeContainer is shown in Algorithm 1. Level i of the tree is implemented using an array tree[i], which allows $O(1)$ access to any node in a level. The root is the only node at level 0. Each node also keeps pointers to its father and children, as well as a bit *side*, indicating whether it is the left or right child of its father.

3.1.1 Task Insertion

Tasks are inserted in a tree using the $put()$ operation. First, put finds a free node to insert the task. Then it updates the presence bits of the node's ancestors. Because a tree has bounded size, task insertions can fail if they do not find a free node in the tree. Below, we describe the main steps in a put.

Algorithm 1 TreesContainer, a scalable bounded task pool algorithm.

```
 1: TreeNode data structure:
        ▷ ver: version of the metadata
        ▷ p indicates presence of tasks in left/right subtrees
        ▷ ⟨ver, p⟩ is kept by a single AtomicInteger in Java
 2:     [⟨ver, p⟩, ⟨ver, p⟩]: meta
 3:     int: pending
 4:     boolean: isDirty        ▷ true if the node has been already used
 5:     Data: task
 6:     int: side               ▷ 0 for the left child, 1 for the right child
 7: Tree data structure:
        ▷ tree[$i$] keeps an array of size $2^i$ with the nodes of level $i$
 8:     TreeNode[][]: tree

 9: Function hasTasks(node):
10:     if (node.meta[0].p $\vee$ node.meta[1].p)
11:         then return 1
12:     else return (node.task $\neq \perp$) ? 1 : 0

13: Function put(task):
14:     node $\leftarrow$ findNodeForPut(task)
15:     if (node $= \perp$) then return false
16:     updateNodeMetadata(node, 1)
17:     return true

18: Function findNodeForPut(task):
19:     for level $= 0, 1, \ldots$ do
20:         trials $\leftarrow$ (level $<$ height(root)) ? 1 : $k$
21:         for i $= 1, \ldots,$ trials do
22:             node $\leftarrow$ random node in tree[level]
23:             reserved $\leftarrow$ putInNode(node)
24:             if (reserved $\neq \perp$) return reserved
25:     return $\perp$                        ▷ did not succeed in this tree

26: Function putInNode(node, task)
27:     if (node.father $\neq \perp \wedge$ node.father.task $= \perp$)
28:         return putInNode(node.father, task)
29:     if (node.isDirty.CAS(false, true))
30:         node.task $\leftarrow$ task; return node
31:     else return $\perp$

32: Function get()
33:     while(true):
34:         if (hasTasks(root) $= 0$) return $\perp$
35:         node $\leftarrow$ findNodeForGet()
36:         task $\leftarrow$ node.task
37:         if (task $\neq \perp \wedge$ node.task.CAS(task, $\perp$) $=$ false) continue
38:         updateNodeMetadata(node, 0)
39:         if (task $\neq \perp$) return task

40: Function findNodeForGet()
41:     node $\leftarrow$ root
42:     while(true)
43:         if(node.task$\neq\perp \vee$ node.meta[0].p=node.meta[1].p=0)
44:             return node
45:         node $\leftarrow$ random child among those with p $= 1$

46: Function updateNodeMetadata(node, myVal)
47:     trials $\leftarrow$ 0;
48:     while(node.father $\neq \perp$)
            ▷ check if my operation has been eliminated
49:         if (myVal $\neq$ hasTasks(node)) return
50:         fk $\leftarrow$ father.meta[node.side].p
51:         if (fk $\neq$ hasTasks(node) $\vee$ node.pending $> 0$)
52:             trials $\leftarrow$ trials $+1$
53:             if (updateFather(node) $\neq$ success $\wedge$ trials $< 2$)
54:                 continue                 ▷ try again on this node
55:         node $\leftarrow$ node.father; trials $\leftarrow$ 0

56: Function updateFather(node)
57:     node.pending.FetchAndInc()
58:     new $\leftarrow$ old $\leftarrow$ father.meta[node.side]
59:     new.ver $\leftarrow$ new.ver $+1$; new.p $\leftarrow$ hasTasks(node)
60:     success $\leftarrow$ father.meta[node.side].CAS(old, new)
61:     node.pending.FetchAndDec()
62:     return success
```

Finding an unoccupied node. Function *findNodeForPut()* finds a free tree node for task insertion. It iterates over the tree levels starting from the root (lines 19–24). At each level, a random node x is chosen, and the algorithm tries to put the task in the *highest* free node on the path from x to the root. This is done using the recursive function *putInNode()* (lines 27–31). Nodes are reserved by CASing the $isDirty$ flag. Having nodes search for a free ancestor increases put's step complexity from $O(h)$ to $O(h^2)$ for a tree with height h (see Appendix A.3). However, it also creates denser trees with a more balanced node occupation, as we show in Appendix A.1.

If neither x nor its ancestors can be reserved, another random node is checked. At each level except the last one, a single node is checked. The number of nodes checked at the last level is defined by a parameter k, with higher k's resulting in denser trees. In Appendix A.2, we show that in a tree with height h, at least $2^{\frac{k+2}{k+3}h}$ nodes are occupied before a put operation fails, whp.

Updating ancestors' metadata. After a task is inserted in node x, function *updateNodeMetadata()* updates the presence bits of x's ancestors (lines 48–55). At each node the function checks that the metadata of the father is correct. Contention remains low because in the common case, the presence bits of upper level nodes are not updated when a new task is inserted or removed.

Though the general outline of the algorithm is simple, ensuring linearizability, wait-freedom and low contention require special care, as we describe below.

1. *Ensuring linearizability.* A naïve approach to update x's father's metadata could be to first read the old presence bit of x's father (line 50), then calculate whether x's subtree contains tasks (line 58), and finally CAS a new metadata value if the old value is incorrect (line 60). If the CAS fails, the updater retries. Version numbers are attached to the presence bits in order to avoid ABA problems.

Unfortunately, this simple approach can violate linearizability. Consider nodes x, y and z, where y is the right child of x and z is the right child of y. Node y has a task, so that $x.meta[1].p = 1$. There are two concurrent threads, a consumer t_c that removes the task from y and a producer t_p that inserts a task in z. t_c starts updating the metadata of B's father. It reads the right presence bit at x, which is 1, and decides to update it to 0. We then suspend t_c right before it performs its CAS operation. At this time, t_p starts updating the ancestors of z. It first changes $y.meta[1].p$ from 0 to 1, and then checks the right presence bit at x. Since t_c is paused, $x.meta[1].p$ is still 1, and so t_p decides this value is correct, and terminates. Now t_c resumes, and successfully changes $x.meta[1].p$ to 0. This makes future gets think the tree is empty, so that no get will retrieve t_c's task, violating linearizability.

We solve this problem by letting other threads know about concurrent pending updaters. Whenever a thread t plans to change the metadata of x's father, it increments a *pending* counter at x (line 57); after the update, it decrements the counter (line 61). If a concurrent updater sees $x.pending > 0$, it will update x's father's metadata, regardless of its current value (line 51). This, along with the use of version numbers, will cause the pending thread's CAS to later fail.

2. *Limiting the number of CAS failures.* In the simple algorithm described earlier, an updater thread t that fails to CAS the metadata of x's father will retry the update. This makes t's worst case step complexity linear in the tree size, since every thread that successfully performed an operation in x's subtree can cause t's CAS to fail. However, as we show in Section B, it suffices for t to only try to update x's father's metadata *twice* (line 53). The idea is that if t fails two CASes, then some other thread will have already updated x's father's metadata to the correct value.

3. *Producer/consumer elimination.* We have also adopted the elimination technique used in [7] and [1]. Consider a thread t that inserted a new task at a node, and started updating the node's ancestors. Let x and y be two such ancestors, where y is the father of x. In the function $updateNodeMetadata$, t updated y's metadata (on x's side) to 1 while t was still at x. Thus, if t later arrives at y and sees y's x-side metadata is now 0, it means there has been consumer thread that already removed the task t inserted. In this case, t doesn't need to update any more ancestors, and can terminate early (line 49). This optimization improves performance in scenarios where multiple producers and consumers are working on the same tree.

In Appendix A.4, we show that put operations in TreeContainer are wait-free. Intuitively, this is because the tree is bounded, and because a thread only tries two updates per node. If the tree has height h, the put performs $O(h^2)$ steps. We show in Section 4 that our insertions create a balanced tree, whp. Hence, when the tree contains N tasks, the complexity of a put is $O(\log^2 N)$.

3.1.2 Task Retrieval

The *get()* function in TreeContainer runs in a loop (lines 33–39). If there are no tasks in the tree, as indicated by the presence bits at the root, the function returns $\perp$ (line 34). $get()$ first finds a task at a random node to retrieve from using $findNodeForGet()$, and then updates the metadata of the node's ancestors.

Function *findNodeForGet()* searches for a node to get a task from. When it reaches an unoccupied node, it randomly chooses a nonempty subtree to go down. The randomization reduces contention.

A task T is removed from node x by CASing $x.task$ from T to $\perp$ (line 37). If the CAS succeeds, then the metadata of x's ancestors need to be updated. Otherwise, the algorithm starts a new retrieval attempt. Note that if *findNodeForGet()* finds a node x with $x.task = \perp$, it means that another consumer t_c removed

x's task but still hasn't updated x's ancestors. In order to be wait-free, a consumer needs to make sure that it will not arrive to this empty node infinitely many times. Hence, a consumer that arrives at an empty node x updates x's ancestors even though it did not take x's task (line 38). Updating the ancestors is done the same way as after a task insertion, using *updateNodeMetadata()*.

In Section A.5, we show that get operations are wait-free. Intuitively, this is because a get thread t_c can only fail to take a task from a previously occupied node x if some other thread took x's task. Then, t_c updates the metadata on the path to the root, so that t_c does not go down the same path again. The bounded number of nodes in a tree then limits the number of unsuccessful get attempts.

3.2 Combining TreeContainers in a FIFO List

As stated earlier, CAFÉ maintains a linked list of TreeContainers, adding new trees as old ones become full (see Figure 1). Tasks are returned in FIFO order, up to the tree they are inserted into. This guarantees that the maximum number of overtakers in CAFÉ is bounded by the tree size. Therefore, the tree size is a parameter that determines the trade-off between fairness and contention. Using bigger trees, CAFÉ performs more like a TreeContainer, and so has low contention but less fairness. Using smaller trees, CAFÉ performs more like a FIFO list, so there is higher contention but greater fairness.

Basic approach. A simple way to manage a linked list of trees is to keep one pointer (PT) for producers, which references the tree for puts, and another (GT) for consumers, referencing the tree for gets. Whenever the current insertion tree becomes full, PT is moved forward. Whenever no tasks are left in the retrieval tree, GT is moved forward. Old trees are garbage collected automatically in managed memory systems as they become unreachable.

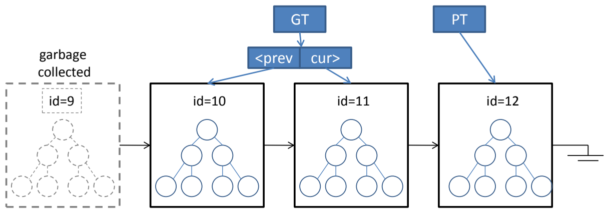


Figure 1: CAFÉ keeps a linked list of scalable task trees. The tree height defines the fairness of the protocol.

This straightforward approach, however, violates correctness, as we now demonstrate. Consider the following scenario. t_p inserts a task in tree T and pauses before changing the metadata of T's root. Consumers assume that T is empty and increment GT to continue to later trees. When t_p finally resumes, we have $GT > PT$, and no consumer will ever retrieve t_p's task.

One way to solve this problem is to reinsert the task in a later tree whenever t_p notices its task may be lost. However, this approach might lead to livelocks, in which producers constantly chase consumers, never finishing their operations. Another method is to maintain a non-zero indicator on each tree (e.g., using SNZI [2]) indicating whether there are concurrent producers working on the tree. But this approach incurs high overhead, for managing both indicators and lists of "pending and active" trees. Our solution is instead based on the idea of moving the consumer pointer GT backwards when a task is added in an old tree.

Managing the list of trees. The pseudo-code for the list of trees pool is shown in Algorithm 2. A put operation tries to insert the task into the tree pointed to by PT (call this tree T). If the insert fails, the algorithm moves to the next tree in the list by incrementing PT (lines 16–17). New trees are created and appended to the end of the list as needed. For reasons we explain later, the pointer for consumers GT actually points to two consecutive trees, $GT.cur$ and $GT.prev$. When an insert succeeds, the producer checks that its task will be retrievable in the future. To this end, it checks that $GT.cur$ does not point to a tree that succeeds T in the linked list (line 13). If it does, the GT pair is moved backwards to $\langle \bot, T \rangle$ in the function $moveGTBack$.

Algorithm 2 CAFÉ algorithm for adjustable fairness and contention.

```
 1: Data structures:
 2:     Node:
 3:         int: id
 4:         ScalableTree: tree
 5: Global variables:
 6:     Node: PT                                   ▷ tree for producers
 7:     $\langle$prev, cur$\rangle$: GT                          ▷ tree for consumers
 8:     int: oldProducers     ▷ for producers that move GT backwards
 9: Function put(task)
10:     while(true)
11:         latest $\leftarrow$ PT
12:         if (latest.tree.put(task) $=$ true) then
13:             if (GT.cur.id $>$ latest.id) moveGTBack(latest)
14:             return
15:         else
16:             if(latest.next $= \perp$) insertNewTree()
17:             PT.CAS(latest, latest.next)
18: insertNewTree()
19:     newNode $\leftarrow$ Node()
20:     cur $\leftarrow$ PT ▷ go to the end of the list
21:     for(; cur.next $\neq \perp$; cur $\leftarrow$ cur.next);
22:     newNode.id $\leftarrow$ cur.id $+1$
23:     cur.next.CAS($\perp$, newNode)          ▷ return even if CAS fails
24: Function moveGTBack(Node: prodTree)
25:     oldProducers.FetchAndInc()
26:     while(true)
27:         gtVal $\leftarrow$ GT
28:         if (gtVal.cur.id $\leq$ latest.id) break
29:         newGT $\leftarrow \langle\perp$, latest$\rangle$
30:         if (GT.CAS(gtVal, newGT) $=$ true) break
31:     oldProducers.FetchAndDec()
32: Function get()
33:     ptVal $\leftarrow$ PT
34:     gtVal $\leftarrow$ GT
35:     while(true)
36:         task $\leftarrow$ gtVal.prev.getTask(); if (task $\neq \perp$) return task
37:         task $\leftarrow$ gtVal.cur.getTask(); if (task $\neq \perp$) return task
            ▷ could not find a task in the tree
38:         if (ptVal.id $\leq$ gtVal.cur.id) return $\perp$
39:         if (oldProducers $= 0$) then
40:             newGT $\leftarrow \langle$gtVal.cur, gtVal.cur.next$\rangle$
41:             GT.CAS(gtVal, newGT)
42:             gtVal $\leftarrow$ GT
43:         else
44:             gtVal $\leftarrow \langle$gtVal.cur, gtVal.cur.next$\rangle$
```

In $moveGTBack$, a producer repeatedly tries to CAS GT to T until a CAS succeeds, or it reads $GT.cur \leq T$. As we want producers to be wait-free, we need to ensure this loop eventually terminates. Thus, we do not allow the GT pointers to move forward while there are pending producers that want to move GT backwards. We increment a counter $oldProducers$ at the start of $moveGTBack$, and decrement it at the end. If a consumer does not find a task in the GT trees, but sees $oldProducers > 0$, it advances to a later tree, but does not increment GT (line 44).

A consumer tries to retrieve a task from the trees pointed to by $GT.prev$ and $GT.cur$ (lines 36–37). If both trees are empty, and if PT points to a later tree than $GT.cur$, then GT is updated to $\langle GT.cur, GT.cur.next\rangle$. This update is performed by first creating a pair with the new tuple values (line 40), and then CASing GT from the old pair to the new one (line 41). Note that the ABA problem does not occur during the CAS, because every newly created pair is a new object whose address is different from the addresses of any old pairs, which are not deallocated throughout the function's execution.

Finally, we explain the reason for using two consumer pointers, $GT.cur$ and $GT.prev$. Suppose GT only pointed to one tree, and consider the following situation. GT and PT both point to a tree T. Producer t_p inserts a new task in T and pauses. Meanwhile, other producers insert new tasks, append new trees and move PT. Suppose a consumer t_c comes to retrieve a task, does not find any tasks in T, and pauses right before changing GT to $T.next$. When t_p resumes, it inserts its task to T, checks that GT is still pointing to T and terminates. When t_c resumes, it changes GT to $T.next$. Now, t_p's task is lost. As we show in the next section, keeping two pointers allows us to solve this problem in a simple and efficient way.

In the next section, we show that both put and get operations in CAFÉ terminate within a finite number of steps with probability 1. Thus, CAFÉ is wait-free.

4 CAFÉ's Properties

In this section, we present the correctness and performance properties of CAFÉ. We only state the main results and describe the ideas behind them, deferring the full proofs to Appendices A and B. For all the results we assume that an adversary controls thread scheduling but cannot influence the randomness threads use. We let h denote the height of a TreeContainer, and k denote the number of insertion attempts in the last layer of TreeContainer (line 20 in Algorithm 1).

4.1 Safety Properties

We start by showing CAFÉ implements a linearizable job pool. Intuitively, if the job pool is nonempty, then a get must be able to find a job. Formally, we prove in Lemma 16 that after any put operation finishes, no subsequent get operation will return $\perp$, until the put's task has been returned. Proving this consists of two parts. First, we prove in Theorem 1 that each TreeContainer CAFÉ uses is itself a linearizable job pool. Second, we show in Lemma 15 that after a put inserts a task in some TreeContainer, subsequent get operations will not skip this TreeContainer when looking for a job. Specifically, we show that $GT.cur \leq PT + 1$.

The key to proving Theorem 1 is Lemma 13, which says that after a put operation has inserted a task in some node of a TreeContainer, $hasTasks(x) = 1$ for every node x on the path from that node to the root of the TreeContainer, until the node's task is removed. We say that the nodes on the path are *marked*. Get operations follow a path of marked nodes, and so will always find a job as long they have not all been removed. We briefly describe the proof of Lemma 13. Let x and y be two nodes a put operation p passes through during $updateNodeMetadata$, where y is the father of x. The invariant we maintain is that the value of $hasTasks(x)$ has been fixed to 1 by the time p starts updating y's metadata. Since p tries to set y's metadata to $hasTasks(x)$, then $hasTasks(y)$ will also be fixed to 1 after p finishes processing y. Thus, all the $hasTasks$ values on the path from p's insertion node to the root will be fixed to 1 inductively.

Next, we briefly describe the proof of Lemma 15. After a put operation has inserted a task in a tree T, it does *moveGTBack* to ensure the value of GT is at most T. There are two ways the put checks this condition. Either it successfully CASed the value $\langle \perp, T \rangle$ into GT, or it read that $GT.cur$ is at most T. Becauses the CASes on GT can be linearized, we can show in the first case that later gets see T (or a smaller value) when they read GT. In the second case, we need to be careful that while the put is checking $GT.cur$ is at most T, there may be a paused get operation, which then increases GT as soon as the put's check finishes. However, even if this happens, $GT.cur$ only moves forward by 1. Since a get operation checks both $GT.cur$ and its preceding tree $GT.prev$, the get will still see the tree that the put inserted into.

The last correctness property we show is that gets return jobs in FIFO order, up to the TreeContainer they were inserted into. This follows simply because jobs are inserted and removed based on the linked list order of the TreeContainers.

4.2 Performance Properties

We first show that our trees are dense: by choosing an appropriate k we can guarantee that a tree with height h is populated with at least $2^{(1-\epsilon)h}$ tasks for an arbitrary $0 < \epsilon < 1$, with high probability. In Appendix A.1, we also show that this density is higher than that achieved by a simple random walk based insertion. More formally, we prove the following lemma in Appendix A.2:

Lemma 1. *In a TreeContainer of height h, if a put operation fails, then the tree contains at least $2^{\frac{k+2}{k+3}\cdot h}$ tasks with probability at least $1 - \frac{1}{2^{(3-\frac{7}{k+3})h+k+1}}$.*

We further demonstrate that TreeContainer performs well under contention. For N concurrent put operations and an arbitrary number of get operations, each put finishes in $O(\log^2 N)$ steps, whp (the proof appears in Appendix A.3):

Lemma 2. *Consider a TreeContainer after N successful* put *operations. Then each of these operations has taken $O(\log^2 N)$ steps with probability at least* $1 - \frac{1}{2(N+1)^{\frac{4}{3}}}$.

We next intuitively demonstrate the wait-freedom of CAFÉ. We first show that put operations are wait-free with probability 1, and then argue that get operations are deterministically wait-free.

A put operation traverses the linked list of TreeContainers until it succeessfully inserts a task in one of them; new TreeContainers are appended if the insertions keep failing. Intuitively, it might seem that this traversal could go on forever. For example, a slow thread t_p could repeatedly try to insert a task in some tree, then pause until all other producers proceed to a new tree, fail its current insert, and have to retry in a new tree. Fortunately, this situation does not happen. Due to the randomness in the algorithm, other threads are likely to have left unoccupied nodes in t_p's tree, which t_p can acquire once it resumes. We formalize this intuition in the following lemma, proven in Appendix A.4.

Lemma 3. *If P producer threads and any number of consumer threads use CAFÉ, then any TreeContainer's* put *operation succeeds with probability at least* $(1 - \frac{1}{2^h})^{k(P-1)} \cdot [1 - (1 - \frac{1}{2^h})^k]$.

Using Lemma 3, we prove the following. Note that CAFÉ using TreeContainers of height 0 is equivalent to a linked list.

Lemma 4. *If the height of TreeContainer is greater than zero, then CAFÉ's* put *operations are wait-free with probability* 1.

In order to show CAFÉ 's get operations are wait-free, we need to show that a consumer does not need to traverse an unbounded number of trees when looking for a task. This is true because each get operation keeps a pointer to the latest TreeContainer when it starts (line 33 in Algorithm 2), and subsequently only checks trees that had tasks before it started. In a linearizable execution, the get is allowed to return $\perp$ when all these trees are empty (in line 38), as all their tasks will have been taken by other gets concurrent with or preceding the current get. We conclude with the following lemma, proven in Appendix A.5.

Lemma 5. *Every get operation of CAFÉ terminates in a finite number of steps.*

5 Evaluation

In this section we evaluate the performance of Java implementation of CAFÉ. We present the compared algorithms in Section 5.1. In Section 5.2 we analyze the performance of the algorithms. Section 5.3 considers the influence of tree height on CAFÉ, and Section 5.4 analyzes its fairness.

5.1 Experiment Setup

We compare the following task pool implementations:

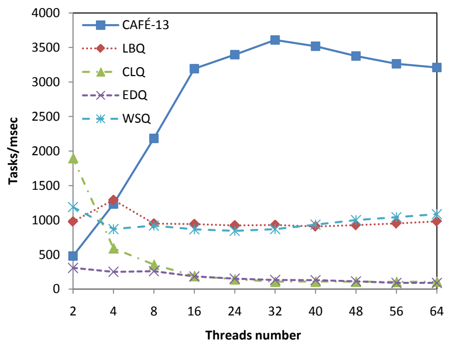
- **CAFÉ-h** – CAFÉ with height h for each tree. Unless stated otherwise, we use $h = 12$.
- **CLQ** – The standard Java 6 implementation of a (FIFO) non-blocking queue by Michael and Scott [6] (class `java.util.concurrent.ConcurrentLinkedQueue`), which is considered to be one of the most efficient non-blocking algorithms in the literature [4, 5].

- **LBQ** – The standard Java 6 implementation of a (FIFO) blocking queue that uses a global reader-writer lock (class `java.util.concurrent.LinkedBlockingQueue`).
- **ED** – The original elimination-diffraction tree implementation [1] (downloaded from the web page of the project), in its default configuration. Tasks are inserted into a diffraction tree with FIFO queues attached to each leaf. The queues are implemented using Java LinkedBlockingQueues. Every tree node contains an elimination array where producers can pass tasks directly to consumers. Changing the tree depth, pool size and spinning behavior did not have a significant effect on the pool's performance. Note that ED trees, like CAFÉ , do not enforce FIFO ordering.
- **WSQ** – Work stealing queues, which do not ensure FIFO. There is one CLQ per consumer. A producer chooses a random queue to insert a task; a consumer tries first to retrieve a task from its own queue, and if it fails, attempts to retrieve a task from the CLQ of another consumer. In the worst case, all consumer queues are checked.

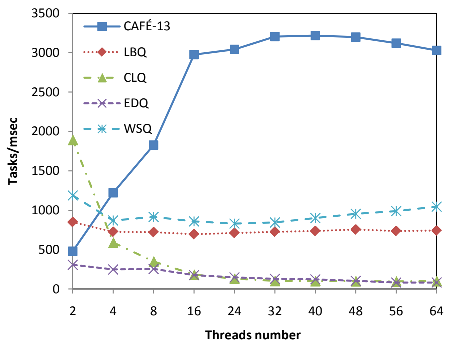
We use a synthetic benchmark for the performance evaluation, in which producer threads work in loops inserting dummy items, and consumer threads work in loops retrieving dummy items.

Unless stated otherwise, tests are run on a dedicated shared memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each processor. JVM is run with the Aggressive-Heap flag on. We run up to 64 threads on the 32 cores. The influence of garbage collection was negligible for all algorithm[5].

5.2 System Throughput



(a) Task insertion rate.



(b) Task removal rate.

Figure 2: Task insertion and retrieval rates (equal numbers of producers and consumers). The throughput of CAFÉ-13 increases up to 32 threads (the number of hardware threads in the system). In this configuration it is $\times 30$ faster than the Michael-Scott ConcurrentLinkedQueue and over three times higher than all other implementations, including the ones not providing FIFO. CAFÉ continues demonstrating high throughput even when the number of threads increases up to 64.

Workloads with the same number of producers and consumers. In Figure 2 we show the average insertion and retrieval rates in a system with an equal number of producers and consumers. Both graphs demonstrate the same behavior. The throughput of CAFÉ increases up to 32 threads, the number of hardware threads in our architecture. At this point, the throughput of CAFÉ is $\times 30$ higher than the Michael-Scott queue or the ED pool. It is also over three times higher than the blocking queue. When the number of working threads exceeds the number of hardware threads in the system, the throughput of CAFÉ decreases moderately, but still outperforms the other algorithms.

[5]This was checked using the verbose:gc flag in JVM.

As we can see in Figure 2, the results of both the Michael-Scott concurrent queue and ED pools are lower than those of other algorithms. This differs from the results demonstrated by Afek *et al.* [1], where ED pools were shown to clearly outperform standard Java queues. This discrepancy seems to follow from differences in the hardware architectures used in our experiments. Afek *et al.* use a Sun UltraSPARC T2 machine with 2 processors of 64 hardware threads each, while in our system there are 8 quad-cores. The difference in architecture is significant due to the *non-uniform* memory access time in multi-processor systems: accessing a memory location from multiple processors is significantly slower than accessing it from multiple hardware threads on the same chip, which usually share a last-level cache. We now show how the non-uniformity of memory accesses influences performance.

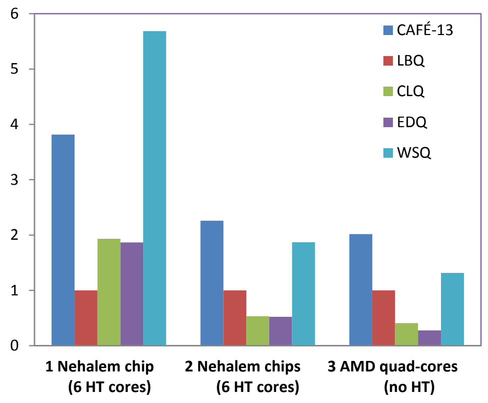
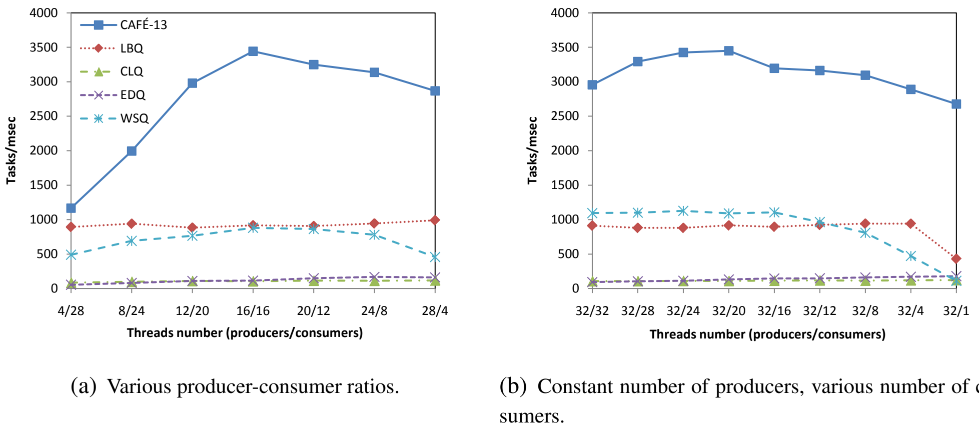


Figure 3: Throughput on different hardware architectures, normalized by the throughput of LBQ. There are 6 producer threads and 6 consumer threads.

Figure 3 demonstrates the throughput of the algorithms in three different configurations: a single Nehalem chip with 6 hyper-thread cores, two Nehalem chips with 6 hyper-thread cores and three AMD quad-cores with no hyper-threading. The algorithms are run with 6 producers and 6 consumers (corresponding to the number of hardware threads available in a single Nehalem chip); the throughput is normalized by the throughput of the Java LinkedBlockingQueue.

We observe that, consistent with the findings of Afek *et al.*, both ED pools and MS non-blocking queue perform twice as well as Java's linked blocking queue when running on a single chip. However, their performances decrease significantly in systems with two or more chips, when memory sharing becomes more expensive. The same is relevant for the work stealing queue, which is the best option when run on a single chip, but behaves worse in multi-chip architectures. Nevertheless, it is worth mentioning that in [1], ED pools achieved the best results when run on many threads (up to 64) on the same core. We were unable to reproduce these results as we do not have an access to a machine with more than 12 HW threads per chip.



(a) Various producer-consumer ratios.

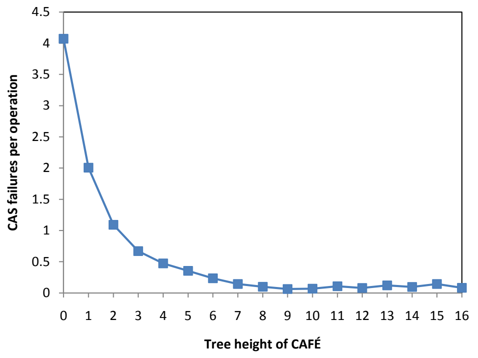
(b) Constant number of producers, various number of consumers.

Figure 4: Task insertion rate for various combinations of producers and consumers. Insertion rate of CAFÉremains significantly higher than that of the competitors for both consumer and producer dominant workloads.

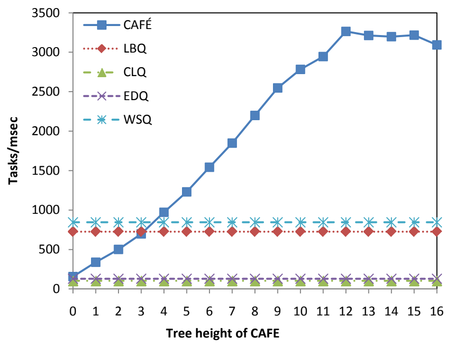
Workloads with different numbers of producers and consumers. In some real-world scenarios, tasks can arrive in bursts and the number of producers varies in time. In such cases, the aim of a task pool is not to delay the producer threads, which typically stay in a critical path. An insertion throughput should remain high even if there are more producers than consumers.

We demonstrate the task insertion rate of the algorithms for the workloads with different numbers of producers and consumers in Figure 4. The insertion rate of CAFÉ is the best when the number of producers is equal to the number of consumers ($16/16$ case in Figure 4(a)). However, it remains high for all other configurations as well, as CAFÉ does not assume any behavioral pattern in order to achieve its performance. This is in contrast to WorkStealingQueue, which needs many consumers in order to disperse the tasks between — its insertion throughput degrades as the number of producers becomes higher than the number of consumers.

5.3 Choosing the Tree Height



(a) CAS failures per operation as a function of tree height.



(b) CAFÉ throughput as a function of tree height.

Figure 5: CAS failures and system throughput as a function of CAFÉ's tree height for 16 producers and 16 consumers. Small trees induce high contention because of linked list manipulations and reduced tree randomization. Excessively large trees induce contention among producers and consumers operating in the same tree.

In Figure 5 we demonstrate CAFÉ's performance for 16 producers and 16 consumers as a function of tree height. Figure 5(a) shows the average number of CAS failures per insertion / removal operation. For height $= 0$, CAFÉ is equivalent to the Michael-Scott concurrent queue, and there are 4 CAS failures per operation. The rate of CAS failures drops quickly for larger trees, becoming less than 0.1 for CAFÉ-8.

The statistics of CAS failures match the throughput graph shown in Figure 5(b). Increasing the tree height improves throughput up to a certain point (12 in our workload), but beyond this performance plateaus. This is because for intermediate tree sizes, producers and consumers usually find themselves in different trees (the latter lagging behind the former), while for heights larger than 13, most of the threads operate in the same tree, which increases contention and decreases performance.

5.4 Fairness

We now show the fairness presented by CAFÉ algorithm. In order to measure fairness we choose to measure the maximal number of tasks that overtake any task in the system (maximal overtakers number). Note that measuring maximal waiting time would be misleading because this number depends on the execution time of each single task, which is not a system parameter.

We did not want our measurements to introduce additional contention points, therefore we check the real-time order only between the tasks inserted by the same producer. We simulate dummy tasks of different length, with ratio of $95\%/5\%$ of short over long tasks. There are 16 producers and 16 consumers in each test.

Figure 6 depicts the maximal number of overtakers of the non-blocking systems. Note that for height h the maximal number of overtakers at CAFÉ is upper bounded by $2^{h+1} - 1$. We see that in practice this number remains much lower, arriving to 600 overtakers for tree height $= 12$. On the opposite, the maximal number of overtakers of WSQ is 2300. This happens because a long running task in the queue-per-consumer pools can block other tasks in its queue, while a large number of short tasks pass in the queues of the other consumers.

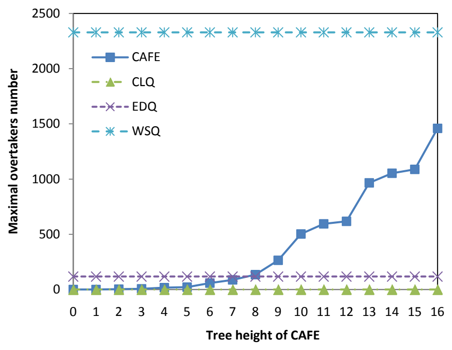


Figure 6: The maximal number of overtakes as a function of tree height.

6 Conclusions

We presented CAFÉ, an efficient wait-free task pool with adjustable fairness and contention. CAFÉ uses a scalable TreeContainer building block, which greatly improves on the performance of queue-based alternatives and provides polylogarithmic step complexity for its put operations. Our evaluations show that CAFÉ significantly outperforms both FIFO and non-FIFO task pool algorithms in multi-chip architectures. As we've seen, existing task pools make different trade-offs between fairness and contention. We believe an interesting theoretical question is whether this trade-off is inherent: is it always more expensive to implement a FIFO queue than an unordered set?

References

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Euro-Par 2010 - Parallel Processing*, pages 151–162. 2010.

[2] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. Snzi: scalable nonzero indicators. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 13–22, 2007.

[3] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, ASPLOS-III, pages 64–75, 1989.

[4] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[5] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. *Distributed Computing*, 20:323–341, 2008.

[6] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, 1996.

[7] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, 2005.

A Analysis

We now formally prove the algorithm properties that were presented in Section 4. Let h_t be the highest level of TreeContainer containing occupied nodes and X_i be the number of occupied nodes of level i of TreeContainer (i varies from 0 to h). In the procedure *findNodeForPut()*, a producer thread tries to reserve a random node while traversing different levels of TreeContainer. Let r_i be the number of unsuccessful trials that a producer can do at level i before it continues to level $i+1$ (lines 21-24 of TreeContainer). Recall that $r_h = k$, i.e., there are k trials at the last level.

A.1 TreeContainer Insertions vs Random Walk

The choice of TreeContainer node for task insertion is determined by *findNodeForPut()* function. The function has two purposes: 1) in order to reduce contention between the threads the function distributes them randomly between different nodes; 2) in order to improve memory utilization and insertion/retrieval latency the function increases the density of occupied nodes.

The straightforward approach for choosing a node for task insertion is a mere random walk (RW) down from the root, where the task is inserted to the first unoccupied node. This simple algorithm achieves low contention, however, as we show in the following lemmas, RW approach yields trees with lower task density.

Claim 1. *If tasks are inserted into TreeContainer by RW, the probability that an insertion increases a current value of h_t is*

$$Pr_{RW}(\textit{increase } h_t) = \frac{X_{h_t}}{2^{h_t}} \tag{1}$$

Proof. Consider the paths from the root to nodes at level h_t. Every path has equal probability to be chosen by RW. Thus, the probability to increase the height equals to the portion of paths that end at occupied nodes at level h_t out of all paths of length h_t. The total number of paths from the root to level h_t equals 2^{h_t}. The number of paths ending at occupied nodes equals X_{h_t}. So, the probability to increase h_t is $\frac{X_{h_t}}{2^{h_t}}$. □

Claim 2. *Assume that* N *tasks have been inserted into TreeContainer by RW. Then the probability that the next insertion by RW increases h_t is at least*

$$Pr_{RW}(\textit{increase } h_t) \leq \frac{N+1}{2^{h_t+1}} \tag{2}$$

Proof. Note that X_i-s have the following constraints:

1. Total number of tasks: $N = \sum_{i=1}^{h} X_i$
2. Tasks, inserted by RW, are structured into a binary tree, therefore $\forall_{0 \leq i \leq h_t - 1} X_{i+1} \leq 2 \cdot X_i$

The constraints ensure that $X_{h_t} \leq \frac{N+1}{2}$ (equality holds in case of a complete tree with a full last level). By Claim 1, we get (2). □

Claim 3. *The probability to increase h_t by inserting a task with TreeContainer* put() *operation is*

$$Pr(\textit{increase } h_t) = (Pr(\textit{increase } h_t \textit{ by RW}))^{r_{h_t}+1} \cdot Pr(\textit{reached level } h_t) \tag{3}$$

Proof. The TreeContainer *put()* function calls to *findNodeForPut()* in order to find a node for task insertion. *findNodeForPut* traverses levels from 0 up to $h_t + 1$. At each level it makes r_i trials to choose a free node uniformly at random. Once an unoccupied node is found, the task is inserted into the highest free predecessor. Such an insertion increases h_t with probability

$$\begin{aligned} Pr(\textit{increase } h_t) = \quad & Pr(\textit{reached level } h_t) \cdot Pr(\textit{failed } r_{h_t} \textit{times at } h_t | \textit{reached level } h_t) \times \\ & \times Pr(\textit{occupy level } h_t + 1 | \textit{failed } r_{h_t} \textit{times at } h_t) \end{aligned}$$

The probability to pick an occupied node at level h_t for r_{h_t} times is

$$Pr(\textit{failed } r_{h_t} \textit{times at } h_t | \textit{reached level } h_t) = (\frac{X_{h_t}}{2^{h_t}})^{r_{h_t}}$$

If all r_{h_t} trials fail at level h_t, then the inserter picks a random node at level $h_t + 1$. The insertion occupies level $h_t + 1$ only if the parent of the chosen node is occupied. The number of nodes at level $h_t + 1$ that have an occupied parent is $2 \cdot X_{h_t}$. Hence, the probability to occupy the level $h_t + 1$ is:

$$Pr(\textit{occupy } h_t + 1 | \textit{failed } r_{h_t} \textit{ times at } h_t) = \frac{2 \cdot X_{h_t}}{2^{h_t+1}} = \frac{X_{h_t}}{2^{h_t}}$$

By Claim 1, $Pr(\textit{increase } h_t \textit{ by RW}) = \frac{X_{h_t}}{2^{h_t}}$, so we get (3). □

Lemma 6. *Insertion by TreeContainer* put() *function always has strictly lower probability to increase the tree height than the insertion by RW.*

Proof. In *findNodeForPut()* procedure of TreeContainer $r_{h_t} = 1$ for $h_t < h$ and $r_{h_t} = k$ for $h_t = h$, so the lemma follows from Claim 3. □

A.2 TreeContainer Density Guaranties

Lemma 7. *Assume N tasks that have been inserted into an empty TreeContainer using* put() *function. Then the probability that the last occupied level h_t does not exceed h_u, $0 < h_u \le h$, is at least* $1 - \frac{(N+1)^{r_{h_u}+2}}{2^{(h_u+1)\cdot(r_{h_u}+1)}}$

Proof. We order the insertions in a run by the time they succeed to occupy nodes in the tree from 1 (first) to N. Let $\{Y_i\}_{i=1}^{N}$ be a set of events, so that Y_i corresponds to the case that insertion number i increases the tree height to $h_u + 1$. The tree height remains h_u after N insertions if none of the these events occurs. According to the *union bound* theorem

$$Pr(\bigcup_{i=1}^{N} Y_i) \le \sum_{i=1}^{N} Pr(Y_i)$$

When insertion i samples tree nodes, there are at most $i - 1$ occupied nodes in the tree. By Claims 2, 3

$$Pr(Y_i) \le (\frac{i+1}{2^{h_u+1}})^{r_{h_u}+1}$$

Thus,

$$Pr(\bigcup_{i=1}^{N} Y_i) \le \sum_{i=1}^{N} (\frac{i+1}{2^{h_u+1}})^{r_{h_u}+1} < \frac{(N+1)^{r_{h_u}+2}}{2^{(h_u+1)\cdot(r_{h_u}+1)}}$$

The lemma follows. □

The following lemma demonstrates the density properties of TreeContainer: it shows that TreeContainer's *put()* operation fails only if most of the nodes in the tree have already been occupied.

Lemma 1 (restated). In a TreeContainer of height h, if a put operation fails, then the tree contains at least $2^{\frac{k+2}{k+3} \cdot h}$ tasks with probability at least $1 - \frac{1}{2^{(3-\frac{7}{k+3}) \cdot h+k+1}}$.

Proof. Recall that at last level of TreeContainer there are k trials to occupy node in *findNodeForPut()* procedure, i.e. $r_h = k$. By Lemma 7, the if $N = 2^{\frac{r_h+2}{r_h+3} \cdot h} = 2^{\frac{k+2}{k+3} \cdot h}$ the probability that all N nodes enter the tree of height h is at least $1 - \frac{2^{\frac{(k+2)^2}{k+3} \cdot h}}{2^{(h+1) \cdot (k+1)}} = 1 - \frac{1}{2^{(3-\frac{7}{k+3}) \cdot h+k+1}}$. □

A.3 TreeContainer Step Complexity

In the current section we investigate the step complexity of TreeContainer's *put()* operations. We say that *step* is either read, write or CAS operation.

Claim 4. *Assume that thread T performs* put() *operation in TreeContainer. If h_t is the last occupied level at the end of this operation, then T has done at most $O({h_t}^2)$ steps in* findNodeForPut() *function.*

Proof. According to the state of the lemma, the last occupied level is h_t, hence *findNodeForPut()* iterates over at most $h_t + 1$ levels. At each level T makes an attempt to reserve a random node v (k attempts at the last tree level). Function *putInNode(v)* is called for each such attempt. During this function T traverses the path from v to the root looking for the highest unoccupied node. If an unoccupied node has been found, T tries to mark its dirty bit using CAS operation. CAS may fail if some concurrent thread has outrun T. In this case, T returns back on the path to v trying to CAS *isDirty* variable of the nodes on the way. If some CAS succeeds, the reservation has done. However, in the worst case, T may fail in *putInNode()* call at every level up to level h_t. Therefore, in the worst case, for every level except h_t, T climbs to the root and goes back with CAS failures. Hence, the number of steps is bounded by $\sum_{i=1}^{h_t} 2 \cdot i \cdot const_i \in O({h_t}^2)$. □

Claim 5. *The step complexity of* updateNodeMetadata(v) *for a node v with height h_v is $O(h_v)$.*

Proof. In function *updateNodeMetada(v)* a thread traverses the nodes on the path from v to the root updating the metadata of some v's predecessors. At each node the thread makes the pre-defined constant number of reads/writes and at most two CAS operations (limited by *trials* variable in lines 52, 53 of TreeContainer). Hence, the total number of steps is $O(h_v)$. □

Lemma 8. *Every* put() *operation of TreeContainer makes at most $O(h^2)$ steps.*

Proof. Function *put()* of TreeContainer performs one call to *findNodeForPut()* and at most one call to *updateNodeMetada()* functions. The lemma follows from Claims 4, 5. □

Lemma 2 (restated). Consider a TreeContainer after N successful *put* operations. Then each of these operations has taken $O(log^2(N))$ steps with probability at least $1 - \frac{1}{2 \cdot (N+1)^{\frac{4}{3}}}$.

Proof. According to Lemma 1, the height of the tree constructed by N insertions is bounded by h_u with probability at least $1 - \frac{(N+1)^{r_{h_u}+2}}{2^{(h_u+1) \cdot (r_{h_u}+1)}}$. At all levels except the last one there are $r_{h_u} = 1$ trials to reserve unoccupied node. If we take $h_u \triangleq \frac{4}{3} \cdot log_2(N)$, by Lemma 1 the last occupied level is at most $\frac{4}{3} \cdot log_2(N)$ with probability at least $1 - \frac{1}{2 \cdot (N+1)^{\frac{4}{3}}}$.

Function *put()* of TreeContainer performs one call to *findNodeForPut()* and at most one call to *updateNodeMetada()*. If h_t is bounded by $\frac{4}{3} \cdot log_2(N)$ at the end of each of N *put()* operations, by Claims 4, 5 each of these operations makes at most $O(log^2(N))$ steps. □

A.4 Probabilistic Wait Freedom of Producers

Lemma 3 (restated). If P producer threads and any number of consumer threads use CAFÉ, then any TreeContainer's *put* operation succeeds with probability at least $(1-\frac{1}{2^h})^{k\cdot(P-1)}\cdot[1-(1-\frac{1}{2^h})^k]$.

Proof. Consider a *put()* operation by some thread T. Assume that an adversarial scheduler tries to fail T's operation. By CAFÉ algorithm, T reads the latest tree from PT (let C denote this tree) and runs TreeContainer's *put()* operation on C. T's operation fails if T does not succeed to reserve an unoccupied node in C. The optimal strategy for the adversary to cause the failure is to suspend T and let other threads occupy the nodes of C.

TreeContainer's *put* operation can fail even if there are unoccupied nodes in C. This happens if there are occupied nodes at level h and *findNodeForPut()* picks k times one of this nodes (TreeContainer lines 21-24). If some thread fails to put a task into C, this thread runs its following *put()* operations on the next trees in the CAFÉ linked list and, thus, cannot affect T's operation anymore.

Let S_T be an event of T success to put a task in C. Let E_i be an event when all the threads except T (T has been suspended by the adversary) have failed to insert a task in C and that there are exactly 2^h-i unoccupied nodes at level h. After E_i occurs, the adversary cannot affect T's operation anymore, because all threads except T stop working on C and move to next trees. Note that E_i are disjoint events and have the property $Pr(\bigcup_{i=1}^{2^h}E_i)=\sum_{i=1}^{2^h}Pr(E_i)=1$. Hence, we can write

$$Pr(S_T)=\sum_{i=1}^{2^h}Pr(S_T\bigcap E_i)=\sum_{i=1}^{2^h}Pr(S_T|E_i)\cdot Pr(E_i)$$

T makes k trials to reserve a uniformly random node at level h, therefore, for $1\le i<2^h$, $Pr(S_T|E_i)=1-Pr(\neg S_T|E_i)=1-(\frac{i}{2^h})^k$; for $i=2^h$, $Pr(S_T|E_{2^h})=0$ (all nodes of C are occupied). Thus,

$$Pr(S_T)=\sum_{i=1}^{2^h-1}[1-(\frac{i}{2^h})^k]\cdot Pr(E_i)\ge[1-(\frac{2^h-1}{2^h})^k]\cdot\sum_{i=1}^{2^h-1}Pr(E_i)$$

As $\sum_{i=1}^{2^h}Pr(E_i)=1$,

$$Pr(S_T)\ge[1-(\frac{2^h-1}{2^h})^k]\cdot(1-Pr(E_{2^h}))\tag{4}$$

Let A_j be the event corresponding to the situation at which exactly j of $P-1$ adversarial threads have failed to put a task in C before 2^h-1 nodes of level h become occupied. Note that events $\{A_j\}_{j=0}^{P-1}$ are disjoint and have the property $Pr(\bigcup_{j=0}^{P-1}A_j)=\sum_{j=0}^{P-1}Pr(A_j)=1$, so we can write

$$Pr(E_{2^h})=\sum_{j=0}^{P-1}Pr(E_{2^h}\bigcap A_j)=\sum_{j=0}^{P-1}Pr(E_{2^h}|A_j)\cdot Pr(A_j)$$

For $j<P-1$, the probability that at least one of $P-j$ adversarial threads succeeds to occupy the last unoccupied node in the following *put* operation is $Pr(E_{2^h}|A_j)=1-(\frac{2^h-1}{2^h})^{k\cdot(P-j)}$. For $j=P-1$, $Pr(E_{2^h}|A_{P-1})=0$, because $Pr(E_{2^h}\bigcap A_{P_1})=0$. Hence, we get

$$Pr(E_{2^h})=\sum_{j=1}^{P-2}[(1-(1-\frac{1}{2^h})^{k\cdot(P-j)})\cdot Pr(A_j)]\le[1-(1-\frac{1}{2^h})^{k\cdot(P-1)}]\cdot\sum_{j=0}^{t-2}Pr(A_j)\le 1-(1-\frac{1}{2^h})^{k\cdot(P-1)}$$

By substituting the inequality result into (4), we finish the proof. $\square$

Lemma 9. *Every producer thread makes a finite number of steps during the function* moveGTBack() *of CAFÉ algorithm.*

Proof. Let T be a producer thread that has called to *moveGTBack()*. Let t_1 be the moment of time at which T has finished *fetchAndInc()* (line 25), and t_2 be the moment of time at which T performs a call to *fetchAndDec()* (line 31) (if this never happens, $t_2 = \infty$).

The consumer threads that start *get()* operations in the interval (t_1, t_2) do not update GT because they do not satisfy the **if** condition of line 39.

There may be consumer threads that have started and not terminated *get()* operations before t_1. We call such consumers t_1-active. A number of these consumers is bounded by the number of threads t using CAFÉ. t_1-active consumer may update GT at most once in the interval (t_1, t_2), because in its next loop iteration (lines 35-44), starting after t_1, it cannot satisfy **if** condition at line 39. Hence, t_1-active consumers can update GT at most t times in (t_1, t_2) interval.

Note that $GT.curr.id$ never exceeds $PT.id$. $PT.id$ only increases and, according to line 38, a consumer never increases $GT.curr.id$ beyond the value of $ptVal.id$, which is read at the beginning of its *get()* operation (line 33).

Let id_{t_1} be the value of $PT.id$ at the moment t_1. Note that $GT.curr.id \le id_{t_1}$. t_1-active consumers execute at most t GT updates and, thus, can increase $GT.curr.id$ up to $id_{t_1} + t$ value.

The producers that start *put()* operations after t_1 insert tasks into nodes with $id \ge id_{t1}$. The producers, which put tasks into nodes with $id \in [id_{t1}, id_{t_1} + t - 1]$, may discover that $GT.curr.id > latest.id$ (line 13) and, as a result, try to move GT backward. However, GT can be moved backward at most t times, because after at most t updates of GT by t_1-active consumers, no consumers move GT forward anymore. Hence, producers that start *put()* operation during (t_1, t_2) interval make at most t updates on GT.

Consider now the producers that have started and have not completed their *put()* operations before t_1. The number of such producers is bounded by the number t of threads that use CAFÉ simultaneously and each of these threads makes at most one update of GT (line 30).

To summarize, there are three types of threads that can update GT concurrently with T, but all these threads can do at most $3 \cdot t$ updates. Therefore, after at most $3 \cdot t$ trials, T succeeds in CAS at line 30 and terminates *moveGTBack()* call. □

Lemma 4 (restated). If the height of TreeContainer is greater than zero, then CAFÉ's *put* operations are wait free with probability 1.

Proof. CAFÉ *put()* operation has two stages: 1) it invokes TreeContainer's *put()* operation on the TreeContainer objects until the first success; 2) it then can possibly call to the function *moveGTBack()*.

By Lemma 8, every TreeContainer's put operation takes $O(h^2)$ steps. By Lemma 3, if $h > 0$, every TreeContainer's *put()* operation succeeds with non-zero probability. Therefore, the first stage terminates in a finite number of steps with probability 1. By Lemma 9, the second stage terminates in a finite number of steps, which finishes the proof. □

A.5 Consumers Wait Freedom

We now show that get operations of CAFÉ are wait-free.

Lemma 10. *After $O(h \cdot 2^h)$ steps without concurrent updates on a TreeContainer by producers every TreeContainer* get() *operation terminates.*

Proof. We say that a node in TreeContainer is reachable from the root, if for every node on the path to the root parent node has predicate 1 in metadata describing the child node. As there are only consumer threads running and their updates write only 0 value to metadata predicates, number of reachable nodes can only decrease.

Consider the function *findNodeForGet()* of TreeContainer. As the function looks for a node by metadata predicates, it always returns a node that is reachable at the moment when the function call has started.

According to *findNodeForGet()* (line 43), it can return either the node with a task, or a node with predicates 0 for both of its children.

In the first case: if a consumer thread succeeds to take the task from the returned node at line 37, the *get()* operation terminates after a call to *updateNodeMetada()*; otherwise some other consumer thread has already taken the task. In both cases, the task is taken from the node and the same node can be returned by the following calls only in the context of the second case.

In the second case: the returned node does not have a task, so *updateNodeMetadata()* is called and the returned node becomes unreachable. As a result, this node cannot be returned in the following calls to *findNodeForGet()*.

Therefore, after at most $2 \cdot (2^{h+1} - 1)$ (twice the size of TreeContainer) invocations of *findNodeForGet()* TreeContainer does not have reachable nodes.

The *get()* function of TreeContainer runs a while loop, at which every iteration makes a single call to *findNodeForGet()* and at most one call to *updateNodeMetadata()* function. If at the beginning of a loop iteration there are no reachable nodes, the *get()* operation terminates (line 34 of TreeContainer). Therefore, there are at most $2 \cdot (2^{h+1} - 1)$ loop iterations.

findNodeForGet() traverses only one path from the root to some inner node and thus makes $O(h)$ steps — by Lemma 5, updateNodeMetada() makes at most $O(h)$ steps. Hence, the total number of steps during the *get()* operation is $O(h \cdot 2^h)$. □

Lemma 11. *Every* get() *operation of TreeContainer terminates in a finite number of steps.*

Proof. According to TreeContainer, a node can be occupied only once. Hence, there is a finite number of *put()* operations that succeed to find an unoccupied node. The number of these operations is bounded by the number of nodes in TreeContainer and, by Lemma 8, every such operation makes $O(h^2)$ updates. Thus, the number of producer updates is bounded by $const_p \cdot h^2 \cdot 2^h$ for some constant $const_p$.

By Lemma 10, there exists a constant $const_c$, s.t. after at most $const_c \cdot h \cdot 2^h$ steps without concurrent updates by producers a *get()* operation of TreeContainer terminates. As we noted before, producers can update TreeContainer at most $const_p \cdot h^{\cdot} 2^h$ times, therefore every TreeContainer's *get()* operation terminates after at most $const_c \cdot h \cdot 2^h \cdot (const_p \cdot h^2 \cdot 2^h + 1)$ steps. □

Lemma 5 (restated). Every get operation of CAFÉ terminates in a finite number of steps.

Proof. Let T_c be a consumer thread performing *get()* operation. Let $gtFirst$ be a pointer to the first tree inserted into the linked list of TreeContainers. Let $const_s$ be a bound on the number of steps taken by a *get()* operation of TreeContainer (by Lemma 11, such a bound exists).

At the beginning of *get()* operation, T_c stores PT and GT values in $ptVal$ and $gtVal$ respectively.

If $(ptVal.id \leq gtVal.curr.id)$, then T_c performs at most two *get()* operations of TreeContainer and terminates.

Otherwise, T_c has to execute *get()* operation on all the trees between $gtVal.curr$ and $ptVal$ in the linked list of trees. Meanwhile, producers can move GT back up to $firstGt$. However, if starting from some moment T_c can take $(ptVal.id - firstGt.curr.id) \cdot 2 \cdot const_s$ steps with no producer threads concurrently moving GT, T_c terminates its *get()* operation.

Note that the producers, which start their *put()* operations after the moment T_c reads PT (line 33), do not move GT before T_c's $ptVal$. Thus, these producers do not affect the termination of T_c *get()* operation. Hence, the only producers that can affect T_c are the producers that started and have not terminated before T_c has read PT. The number of such producers is bounded by the number t of threads using CAFÉ, so, the number of times that GT can be moved back during T_c's *get* operation is t. Therefore, after $t \cdot (ptVal.id - firstGt.curr.id) \cdot 2 \cdot const_s$ steps of T_c, either T_c terminates or producers move GT t times. Hence, after at most $(t+1) \cdot (ptVal.id - firstGt.curr.id) \cdot const_s$ steps, T_c terminates its *get()* operation. □

B Safety Proofs

In this section, we show that TreeContainer and CAFÉ implement linearizable job pools. We will refer to line number r in Algorithms 1 and 2 by $\langle Tr \rangle$ and $\langle Cr \rangle$, resp.

B.1 Safety of TreeContainer

We first show that a single TreeContainer implements a linearizable job pool. That is, a get operation on a TreeContainer only returns $\perp$ if the tasks of all put operations that completed before the get have been taken. The basic idea, stated in Lemma 13, is that between the time when a put operation finished inserting a task in a node and when the task is removed, the entire path between the node and the root is marked. This implies that as long as there are some unremoved tasks in the TreeContainer, there is a marked path which leads get operations to a task.

We first prove a lemma which says that each CAS on a node's metadata sees the value of the previous CAS. This allows processes to correctly transfer information to each other.

Lemma 12. *Let T be any TreeContainer, let $x \in T$ be any node, and let y be x's father. Consider the sequence of successful CASes (at $\langle T60 \rangle$) $C_1, \ldots, C_m$ on $y.meta[x.side]$, and let $R_1, \ldots, R_m$ be the corresponding reads (at $\langle T58 \rangle$) preceding each such CAS. Then for $i = 2, \ldots, m$, R_i occurs after C_{i-1}.*

Proof. From $\langle T59 \rangle$ and $\langle T60 \rangle$, we see that each successful CAS on $y.meta[x.side]$ increments the version number $y.meta[x.side].ver$. Thus, if some R_i occurred before C_{i-1}, the value $y.meta[x.side].ver$ R_i read would be less than $y.meta[x.side].ver$ after C_{i-1}, and so C_i would fail. This is a contradiction. □

In the following, given a node x and a time t, we write $hasTasks(x)$ *at* t to mean the value that the function **hasTasks**(x) would return if evaluated at time t.

Lemma 13. *Consider any TreeContainer T, and let p be a completed put operation that inserted a task in node $x_0 \in T$. Suppose that by some time τ, no get operation has removed the task from x_0,* i.e. $\langle T37 \rangle$ *with $node = x_0$ has not occurred. Then for every node x on the path from x_0 to the root of T, $hasTasks(x) = 1$ at τ.*

Proof. We first show the lemma holds for x_0. Since p finished its operation, it inserted a task in x_0 at $\langle T30 \rangle$ at some time $t < \tau$. Also, since a get for x_0 has not occurred by τ, then $x_0.tasks \neq \perp$ from t to τ. So $hasTasks(x) = 1$ at τ.

To show the lemma for the other nodes on the path, we first prove the following claim, which says that if p passes through some node whose $hasTasks = 1$ from some point onwards, then p will also pass through the node's father, and the father's $hasTasks = 1$ from some point onwards.

Claim 6. *Let x be a node on the path from x_0 to the root of T, let y be x's father. Suppose there's a time $t < \tau$ such that $p.node = x$, and also $hasTasks(x) = 1$ from t until τ. Then there's another time $t' > t$ such that $p.node = y$, and $hasTasks(y) = 1$ from t' until τ.*

Proof. Since p finished its operation by time τ, then it returned either from $\langle T48\rangle$ or $\langle T49\rangle$. Also, because p is a put operation, $p.myVal = 1$. So since $p.node = x$ at t and $hasTasks(x) = 1 = p.myVal$ from t till τ, then p did not return from $\langle T49\rangle$ while $p.node = x$. Thus, p does $\langle T55\rangle$ at some time $t' > t$, and the first part of the claim holds.

To show the second part, consider 3 cases. Either the if on $\langle T51\rangle$ was false, or the if on $\langle T51\rangle$ was true and the if on $\langle T53\rangle$ was true at some time, or the if on $\langle T51\rangle$ was true and the if on $\langle T53\rangle$ was always false. In the first case, let t'' be the time when p did $\langle T51\rangle$. Then $t < t'' < t'$, and so $hasTasks(x) = 1 = y.meta[x.side].h$ at t'', and $x.pending = 0$ at t''. Consider any step which changes the value of $y.meta[x.side].h$ between t'' and τ. By inspection, we see that this must be a successful CAS step C on $\langle T60\rangle$, say by process q. Let R be the read on $\langle T58\rangle$ by q before C. Then R must have occurred after t'', because if R occurred before t'', then q also did $\langle T57\rangle$ before t'', and did $\langle T61\rangle$ after t'', so that $x.pending > 0$ at t'', which is a contradiction. Since R occurred at $t'' > t$, then $new.h = hasTasks(x) = 1$ when q performs $\langle T59\rangle$, and so after C, $y.meta[x.side].h = new.h = 1$.

In the second case, let t'' be the time when p did a successful CAS to $y.meta[x.side]$ in $\langle T60\rangle$ during one of its executions of **updateFather**. Since $hasTasks(x) = 1$ from t to t'', then p CASed the value 1 into $y.meta[x.side].h$, and so $y.meta[x.side].h = 1$ right after t''. Consider any (CAS) step C by a process q that changes the value of $y.meta[x.side]$ between t'' and τ, and let R be the read at $\langle T58\rangle$ by q right before C. Then R occurs after t'', by Lemma 12. So R reads $hasTasks(x) = 1$ sometime after $t'' > t$, and after C, we have $y.meta[x.side].h = 1$.

In the last case, let C_1, C_2 be the two failed CASes on $\langle T60\rangle$ that p performed during its 2 executions of **updateFather**, and let R_1, R_2 be the reads on $\langle T58\rangle$ preceding C_1, C_2, resp. Since C_1 failed, then there must be a successful CAS C_1' which occurred between R_1 and C_1; otherwise, C_1 would have succeeded. Similarly, there must be a successful CAS C_2' by a process q between R_2 and C_2. Let R_2' be the read at $\langle T58\rangle$ by q before C_2'. Then by Lemma 12, R_2' occurs after C_1'. So, since R_1 occurs after t, and R_2' occurs after C_1' which occurs after R_1, then q sees $hasTasks(x) = 1$ when it does $\langle T59\rangle$ after R_2'. Thus, $y.meta[x.side].h = 1$ after C_2'.

The above 3 cases show that $y.meta[x.side].h = 1$ from t' until τ, and so $hasTasks(y) \geq y.meta[x.side].h = 1$ from t' until τ. □

We now complete the proof of the lemma. As stated earlier, $hasTasks(x_0) = 1$ from some $t < \tau$ to τ. Then by repeatedly applying Claim 6, we have $hasTasks(x) = 1$ for every x on the path from x_0 to T's root, from some time before τ until τ. So the lemma holds. □ □

Corollary 1. *Let T be a TreeContainer, and suppose a get operation g performs $T.getTask()$ at time t_1, and returns $\perp$ at time t_2. Let p be a task that was put into T before t_1. Then p was removed from T before t_2.*

Proof. Suppose for contradiction that p's task was not removed before t_2. Let x be the node that p's task was inserted into. Then by Lemma 13, every node x' on the path from x to the root of T has $hasTasks(x') = 1$ during $[t_1, t_2]$. Thus, g does not return $\perp$ in $\langle T34\rangle$. g also doesn't return $\perp$ in $\langle T39\rangle$, as this would imply that some other get successfully took x's task in $\langle T37\rangle$. Hence, g does not return $\perp$, which is a contradiction. □

Theorem 1. *TreeContainer implements a linearizable producer-consumer pool.*

Proof. From $\langle T37\rangle$, we see that every task can be returned by at most one task. Also, Corollary 1 shows that a get operation only returns $\perp$ if all the put operations that finished before it started have been returned. So, the theorem follows. □

B.2 Safety of CAFÉ

We now show that CAFÉ implements a linearizable job pool. Since CAFÉ uses a list of linearizable TreeContainers, the main property we show, in Lemma 15, is that if there is an unremoved job in some TreeContainer T, then a get operation will start looking for jobs starting from T or an earlier TreeContainer. This implies that if a get operation returns $\perp$, then all the jobs that were put into some TreeContainer before the get started have been removed. This implies CAFÉ is linearizable.

We begin by showing that processes can correctly pass information to each other by performing CAS on GT. Note that GT is only changed by a CAS at either $\langle C30\rangle$ or $\langle C41\rangle$. In each case, there's a preceding read on GT, at either $\langle C27\rangle$ for a CAS at $\langle C30\rangle$, or $\langle C34\rangle$ or $\langle C42\rangle$ for a CAS at $\langle C41\rangle$. Given an execution of CAFÉ , denote the sequence of CASes on GT by $C_1, \ldots, C_m$, and denote the sequence of corresponding reads by $R_1, \ldots, R_m$.

Lemma 14. *For $i = 2, \ldots, m$, R_i occurs after C_{i-1}.*

Proof. Each time a CAS occurs, we try to set GT to a new value created on $\langle C29\rangle$ or $\langle C40\rangle$. As mentioned earlier, since CAFÉ is implemented in Java, each of the new values is a Java reference, and hence, is *unique* for the entire execution of CAFÉ . Thus, if R_i occurs before C_{i-1}, for some i, then C_{i-1} will change the value of GT that R_i read, and hence C_i will fail, which is a contradiction. □

Lemma 15. *Let p be a completed put operation that inserted a task in TreeContainer T. Suppose at some time τ, p's task has not been removed. Then $GT.cur.id \leq T.id + 1$ at τ.*

Proof. We will prove a stronger statement than the lemma. Define a time t as follows. Since p completed its operation, it returned from **moveGTBack** either at $\langle C28\rangle$ or $\langle C30\rangle$. In the first case, let t be when p did $\langle C27\rangle$, and in the second case, let t be when p did $\langle C30\rangle$. The lemma follows from the following claim.

Claim 7. $GT.cur.id \leq T.id + 1$ *from t until τ.*

Proof. Consider the sequence of successful CAS operations on GT between t and τ. Note that these are the only operations that change GT's value. We prove the claim using induction on the sequence of CASes.

The base case is time t, when no CASes have occurred yet. If t is defined as in the first case above, then $GT.cur.id = gtVal.cur.id \leq T.id$ at t. If t is defined as in the second case, then after p's CAS at $\langle C30\rangle$, we also have $GT.cur.id \leq T.id$.

Next, assume inductively that $GT.cur.id \leq T.id + 1$ after some number of CASes on GT. We show the condition still holds after the next CAS. This CAS occurs at either $\langle C30\rangle$ or $\langle C41\rangle$. If the CAS occurs at $\langle C30\rangle$, then the corresponding read occurred at $\langle C27\rangle$. By induction, we have $gtVal.cur.id = GT.cur.id \leq T.id{+}1$ from this read. Since $\langle C30\rangle$ occurred, the if in $\langle C28\rangle$ was false, and so $gtVal.cur.id > latest.id$. Then, the CAS on $\langle C30\rangle$ sets $GT.cur.id$ to $latest.id < gtVal.cur.id \leq T.id + 1$, and so $GT.cur.id \leq T.id + 1$ after the CAS.

In the other case, the CAS occurs at $\langle C41\rangle$. We claim that at most one such CAS occurs after t, and $GT.cur.id \leq T.id + 1$ after this CAS. Let C be the last successful CAS on GT before t, and C' be the first successful CAS on GT after t. Then by Lemma 14, the read operation R' on GT corresponding to C' occurs after C. Since $GT.cur.id \leq T.id$ immediately after t, and C was the last CAS to change GT's value before t, then R' read $GT.cur.id \leq T.id$. From $\langle C40\rangle$, we see that C' increased $GT.cur.id$ by 1, and so

$GT.cur.id \leq T.id + 1$ after C', which proves the second part of the (sub)claim. To show that C' is the only successful CAS on GT between t and τ, consider the read R corresponding to any CAS attempt on GT after t, say by a process q. If R occurs before C', then q's CAS will fail, by Lemma 14. Otherwise, R occurs after C', which occurs after t, which occurs after p finished $\langle C12 \rangle$. If $gtVal$ is the value of GT that R read, then by induction, we have $gtVal.prev.id = T.id$ or $gtVal.cur.id = T.id$. Thus, q will do $T.getTask()$ either in $\langle C36 \rangle$ or $\langle C37 \rangle$. Since p's task has not been removed by time τ, then by Corollary 1, $T.getTask() \neq \perp$. Thus, q will not advance past $\langle C37 \rangle$, and will not do a CAS on GT. This shows that C' is the last CAS on GT from t till τ. □

□

Lemma 16. *Suppose a get operation g in CAFÉ returns $\perp$ at a time t. Then for every put operation p that completed before the start of g, p's task was removed by some get operation before t.*

Proof. Suppose for contradiction there is some p that finished inserting a task in a tree T, and the task was not removed before t. Then by Lemma 15, we have $GT.cur.id \leq T.id + 1$ when g does $\langle C33 \rangle$ and $\langle C34 \rangle$. Thus, in some iteration of g's while loop at $\langle C35 \rangle$, g does $T.getTask()$. By Corollary 1, $T.getTask() \neq \perp$, and so g does not return $\perp$, which is a contradiction. □

Theorem 2. *CAFE implements a linearizable producer-consumer pool.*

Proof. Clearly, each get operation only returns puts that start before the get finishes. Lemma 16 shows that a get only returns $\perp$ if all the puts that finished before it started are taken by other gets. Together, these define the semantics of a producer-consumer pool. □

Lemma 17. *Let p_1, p_2 be two put operations inserting into trees T_1, T_2, resp., with $T_1.id < T_2.id$. Suppose that p_1 and p_2's tasks are not removed at time t_1. Then if p_1 and p_2's tasks are both removed by gets that start after t_1, p_1 is removed before p_2.*

Proof. Let t_2 be the first time when either p_1 or p_2's task is removed. Then by Lemma 15, at any time between $t \in [t_1, t_2]$, we have $GT.id \leq T_1.id + 1 \leq T_2.id$. Let g be any get operation that starts after t_1. Then from $\langle C36 \rangle$ and $\langle C37 \rangle$, g does $T_1.getTask()$ before $T_2.getTask()$. Also, by Corollary 1, up to time τ_1, g's $T_1.getTask()$ does not return $\perp$. Thus, p_1's task is removed before p_2's. □