CAFÉ: Scalable Task Pools with Adjustable Fairness and Contention

Dmitry Basin

CAFÉ: Scalable Task Pools with Adjustable Fairness and Contention

Research Thesis

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Electrical Engineering

DMITRY BASIN

Submitted to the Senate of the Technion – Israel Institute of Technology
CHESHVAN 5772 HAIFA OCTOBER 2011

The Research Was Done Under the Supervision of Prof. Idit Keidar from the Department of Electrical Engineering, Technion, in the Department of Electrical Engineering, Technion. It was accepted to the Distributed Computing (DISC) 2011 Conference [3].

The Work was Done in Cooperation with Rui Fan, Idit Keidar, Ofer Kiselov and Dmitri Perelman.

THE GENEROUS FINANCIAL HELP OF THE TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY IS GRATEFULLY ACKNOWLEDGED

Acknowledgments

I would like to thank the following people for their help, guidance and support during my studies for the master degree. My advisor Prof. Idit Keidar, for revealing to me the distributed and parallel systems area of study, for very friendly atmosphere for work, wise advices and smart guidance. I thank Dmitri Perelman for saying one day: “Let’s do the master degree!”, for great friendship, guidance and support during this work. Thanks to my family: parents Michael and Svetlana; brother Pavel and sister Nina for believing in me and supporting in all difficult moments. I am also very grateful to my girlfriend, Anastasia, for encouragement and having a great time despite of the efforts required by this work.

Contents

List of Figures

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.

TreeContainer scalability of is shown 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. A further proof shows 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.

Chapter 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.Task pools are also called producer-consumer pools ; producers perform puts, and consumers gets. Pools have a number of important applications in multiprocessor computing, such as maintaining the set of pending tasks in a parallel computation. For example, there is a highly concurrent web server, SEDA [13], that uses such task pools as a building block. A key challenge in these applications is to ensure that 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 times a task is *overtaken* by another task, i.e., is retrieved after a task that was inserted later than it.

More formally, a task pool is a *concurrent object*, which resides in shared memory, and multiple threads perform operations on it concurrently. We are interested in implementing an *atomic* or *linearizable* [9] pool. Intuitively, an atomic implementation of a concurrent object allows to think that every operation takes an effect at some moment in time between its invocation and response without influence of concurrent updates from other threads.

Most current task pools implementations [11, 10, 1, 2, 12] provide *lock-freedom* [7] guarantees about execution progress, ensuring that at least one thread progresses in its operation regardless of the performance or failures of other threads. In our work we focus on *wait-free* and *randomized wait-free* pool implementation. Wait-freedom implies that

each thread executing an operation performs a finite number of steps, while randomized wait-free implementation guarantees a finite number of steps with probability 1. Formal definitions of correctness and progress properties appear in Chapter 3.

In this thesis, we present CAFÉ (Contention And Fairness Explorer), an efficient randomized wait-free 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 tasks 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, lock-free 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 use locks, and require puts and gets to wait for each other. In contrast, all operations in our algorithm are wait-free. 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. SegmentedQueue (SQ) of [2] uses, similar to CAFÉ, the idea of managing a linked list of segments. However, the segments of SQ are implemented by simple arrays and, as a result, are much less scalable. In addition, SQ has lock-free liveness guarantees, while CAFÉ is wait-free.

We have implemented CAFÉ in Java, and tested its performance on a 32-core machine[1]. 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 which uses Java's state-of-the-art lock-based queue (even though CAFÉ is wait-free). Also, CAFÉ is over three times faster than SQ and over 30 times faster than ED pools, while providing stronger liveness and fairness guarantees.

The rest of the thesis is organized as follows: Chapter 2 describes background and previous work. Chapter 3 introduces the model of CAFÉ's implementation context and formally defines the α-fair task pool problem it solves. Chapter 4 introduces the CAFÉ algorithm and provides intuition for its correctness and efficiency. Chapter 5 states key lemmas regarding the correctness, liveness, and performance properties of CAFÉ. Chapter 6 empirically evaluates CAFÉ, compares its performance to that of other algorithms, makes performance breakdown, and considers the cost of implementing fairness requirements. Chapters 7– 9 include formal proofs of the lemmas stated in Chapter 5: Chapter 7 deals with correctness, Chapter 8 proves performance properties, and Chapter 9 formally proves CAFÉ's liveness properties. Finally, Chapter 10 concludes the thesis.

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

Chapter 2

Related Work

A common approach to implementing concurrent task pools is to use FIFO queues for task management. The state of the art concurrent FIFO queue implementation is Michael and Scott's queue ($M\&S$) [11]. The queue is represented by an unidirectional linked list. It has global *head* and *tail* references. Every node in the list has a *next* reference to the following node. The last node's next is $\perp$. A put operation adds a new node to the end of the list and updates the tail reference. During the operation, the thread has to perform two CAS instructions: first, update by CAS the next reference of the last node in the list to point to the new node; second, update by CAS the global tail reference to point to the new node. A get operation removes the head node of the list by a single CAS on the global head reference, moving it to point to the following node in the list. The $M\&S$ queue is lock free and linearizable with respect to the sequential specification of a FIFO queue.

Later work by Ladan-Mozes and Shavit [10] improves the $M\&S$ implementation achieving a single CAS per put operation in their *optimistic queue*. The optimistic queue manages a bidirectional linked list where every node has *p*rev and *n*ext references to neighboring nodes. A put operation allocates a new node with *prev* pointing to the current tail node, and CASes *tail* to point to the allocated node. In rare cases, there is a possible inconsistent state of next, prev fields in some nodes, but together with head, tail references threads can discover this and fix the state and, finally, use correct values. The optimistic queue, like $M\&S$, is lock-free and linearizable with respect to a FIFO queue sequential specification.

Both $M\&S$ and optimistic queue have simple implementations and are widely used. However, they do not scale well under high contention, because many threads contend on few memory locations - the head and tail of the queue. Consider N threads performing get operations simultaneously on such a queue (the case of put is symmetric). All the threads compete to CAS a single head reference. Only one thread succeeds in each iteration, and the remaining threads retry the CAS on the new head. The total number of CASes in such a run is $N + (N-1) + (N-2) + ... + 1 = \frac{N \cdot (N+1)}{2}$. Thus, an average thread does $\Omega(N)$ CASes per single get operation. In contrast, in CAFÉ most operations are done inside a single TreeContainer, and we formally prove that, in TreeContainer, in N successful put operations, every thread performs whp $O(log^2(N))$ steps consisting of CAS, read and write instructions. The improvement is possible due to relaxation of strict FIFO requirement.

Further work by Moir et al. [12] tries to solve the scalability problem of a single contention point by adding an elimination array to an $M\&S$-like queue. When the queue is empty, put and get threads can "eliminate" each other at randomly chosen entries of the elimination array without proceeding to the linked list queue operations. So when the queue is empty, producer and consumer threads meet at multiple locations in the elimination array which reduces the average number of CASes a thread has to do per operation. 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. When the queue is not empty, elimination is not used and we get the usual $M\&S$ performance. When concurrency is low, producer/consumer eliminations are rare, and threads waste time waiting for elimination. The queue of Moir et al. [12] is lock-free and linearizable with respect to a FIFO queue sequential specification.

The elimination of [12] cannot be used for a non-empty queue because of FIFO requirements. CAFÉ makes the observation that strict FIFO ordering is not necessary for a task pool, and thereby achieves a scalable algorithm, which performs well under both high and low concurrency, 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 6.2, ED pools do not scale well on multi-chip architectures. We explain this by the fact that all the elimination arrays used in the ED pool are constantly accessed by all threads at random locations. As a result, all the elimination arrays are replicated in the caches of all chips, and very frequently, operations require expensive exits to neighboring chips to maintain cache coherency. Additional disadvantages of ED pool are that, unlike CAFÉ , it is not wait-free, and offers no fairness guarantees.

In [2] Afek et al. present a non-FIFO *segmented queue* (SQ) which, like CAFÉ, has a bound on the number of overtaken tasks. Like CAFÉ, SQ maintains a linked list of segments. Each segment is an array of nodes in the size of the overtakers' bound. Each node has a deleted Boolean marker, which indicates if it has been taken by get. Each producer iterates over the last segment in the linked list in some random permutation order. When it finds an empty cell, it performs a CAS operation attempting to put its new element. In case the entire segment has been scanned and no available cell is found (implying that the segment is full), the operation attempts to add a new segment to the list. The get operation is similar: the consumer iterates over the first segment in the linked list in some random permutation order. When it finds an item that has not yet been taken, it performs a CAS on its deleted marker in order to delete it, and if successful, this item is considered dequeued. In case the entire segment was scanned and all the nodes have already been dequeued (implying that the segment is empty), then it attempts to remove this segment from the linked list and repeats the process on the next segment. If there is no next segment, the queue is considered empty. We can see in Section 6.2 that distributing contention over arrays of segments, indeed, performs better than FIFO implementations. However, it still is at least 3 times worse than CAFÉ's performance. The main reason for this is the inefficient search for required nodes by producers and consumers. When an SQ segment is full producers have to traverse all the array to find a place for item insertion, and when a segment is empty, consumers have to traverse all the array while searching for an item to retrieve. We show in Section 6.3 that SQ performance improves only up to a segment size of 255 and further relaxation of FIFO has no benefit. CAFÉ, on the other hand, uses as its "segment" TreeContainer, which provides much better navigation to empty nodes for producers and to non-taken tasks for consumers. In CAFÉ, producers do not have to

traverse all the nodes of a TreeContainer before adding a new segment. Producers can discover in a few steps that the current segment has few empty nodes and allocate a new TreeContainer. As a side-effect of this approach, producers can work on more than one TreeContainer simultaneously, which requires much more complicated maintenance of TreeContainers than that of SQ, which CAFÉ still does efficiently. Consumers of CAFÉ use metadata presence bits of TreeContainer to navigate efficiently to non-taken tasks. We show in Section 6.2, that CAFÉ's throughput grows up to a size 2^{14} of TreeContainer, providing a good payoff for FIFO relaxation. In addition, CAFÉ's put and get operations are wait-free, while those SQ are only lock-free.

The idea of using concurrent tree-based data structures for reducing contention has appeared in previous works not related to task pools [5, 4]. For example, Scalable NonZero Indicator (SNZI) [4] is a shared object that supports *Arrive* and *Depart* operations, as well as a *Query* operation that returns a boolean value indicating whether the number of completed Arrive operations exceeds the number of completed Depart operations (i.e., whether there is a *surplus* of Arrive operations). The SNZI data structure is organized as a rooted tree of nodes implemented by SNZI objects. Arrive and Depart operations are performed on the nodes of the tree, increasing or decreasing the node's surplus, respectively. Operations preserve the property that the root has surplus if and only if some node in the tree has surplus. The idea behind SNZI is that the child node acts as a filter for its parent updates. For example, when some thread arrives to a child node that already has a surplus, the thread can finish the arrival quickly without updating parent node, because all the predecessors of the child node up to the root already have surplus. SNZI's approach of tracking surplus on nodes resembles our TreeContainer's metadata presence bits. However, TreeContainer supports not only indication about presence of tasks in the tree, but also navigation to tasks, such that any task previously inserted by some producer thread can be taken by any consumer thread. In addition, unlike previous work [5, 4], we prove formal bounds on the worst-case step complexity of our TreeContainer algorithm.

Chapter 3

Model and Problem Definitions

The problem we solve in this thesis is implementing a wait-free linearizable α-fair task pool. In Section 3.1 we describe the model and runtime environment. Then, in Section 3.2, we define the linearizability criterion for concurrent data structures. In Section 3.3, we introduce a sequential specification for α-fair task pool. Finally, in Section 3.4, we consider formal definitions of progress guarantees.

3.1 Implementation Environment

We consider a shared memory environment where execution threads have a shared heap, shared read only code, and separate stack memory spaces. The implementation assumes that objects that are not accessible from any thread stack are automatically garbage collected by the underlying middleware (in our implementation the Java VM). We assume that order dependencies between instructions in different threads are according to the Java Memory Model[6] specification. The scheduler can suspend a thread, for an arbitrary duration of time, at any moment after termination of a basic processor instruction (read, write, CAS, FetchAndInc, FetchAndDec).Threads cannot be suspended in the middle of a basic instruction. In addition, we assume that threads can independently generate uniform random integer values of any range.

3.2 Concurrent Objects, Linearizability

Formally, a task pool is a *concurrent object* [9], which resides in a memory shared among multiple threads. As a concurrent object, it has some state and supports a set of operations. Multiple threads can simultaneously perform operations on the same object. Such operations may update the state of the object. Each thread operation takes time and, thus, has a moment of *invocation* and a moment of *response*. When threads concurrently perform operations on concurrent objects, they generate a *history* [9], which is an ordered list of invocation and response events of concurrent object operations. The order of events is according to the time line in which they occurred. An operation invocation event is represented by the record $O.method_T(args)$, where O is the concurrent object, $method$ is the invoked operation, $args$ are the invocation arguments and T is the thread that started the invocation. An operation response event is represented by the record $O.method_T \rightarrow results$, where *results* are the operation result set. In a given history, we say that a response matches a prior invocation if it has the same object O and thread T, and no other events of T on object O appear between them. A *sequential history* is a history that has the following properties: 1) the first event in the history is an invocation; 2) each invocation, except possibly the last, is immediately followed by a matching response.

Each concurrent object has a *sequential specification* defining which sequential histories are legal. In other words, it actually says what actions are correct when operations are invoked sequentially without overlapping calls from different threads.

For defining correctness criteria of concurrent objects we consider the following definitions. An invocation is *pending* in history H if no matching response follows the invocation. An *extension* of history H is a history constructed by appending zero or more responses matching the pending invocations of H. *Complete(H)* is the sub-sequence of H created by removing all pending invocations of H. $H|T$ is a history consisting of exactly the events of thread T in history H. Two histories H and H' are *equivalent* if for each thread T, $H|T = H'|T$.

For a given sequential specification of a concurrent object, the *linearizability* [9] correctness criterion can be defined. A history H is *linearizable* if it has an extension H' and there is a sequential history S such that:

1. S is legal according to the sequential specification of the object.

2. *Complete(*H'*)* is equivalent to S.

3. If method response m_0 precedes method invocation m_1 in H, then the same is true in S.

Concurrent objects that have only linearizable histories are called *linearizable* or *atomic*. Intuitively, concurrent object is linearizable if it requires each concurrent run of its method calls to be equivalent in some sense to a correct serial run.

3.3 α-Fair Task Pool Sequential Specification

An α-fair task pool supports $put(t)$ and $get() \rightarrow t$ operations, where t is a task or $\perp$. We assume that tasks inserted into the pool are unique. That is, if $put(t)$, and $put(t')$ are two different invocations on a task pool, then $t \neq t'$. This assumption is made to simplify the definitions, and could be easily enforced in practice by tagging tasks with process ids and sequence numbers. An α-fair task pool satisfies the following properties:

1. Get validity: get returns a task t previously inserted by a $put(t)$ operation, and not returned by any preceding get operation; if there is no such task, it returns $\perp$.

2. α-Fairness: if between the response of $put(t)$ and the invocation of $put(t')$, α put operations have started and terminated, then no get operation can return t' before t has been returned.

3.4 Concurrent Object Liveness Properties

Threads may invoke a concurrent object's operations simultaneously. A concurrent object implementation is *lock-free* if there is guaranteed system-wide progress, i.e., at least one thread always makes progress in its operation execution, regardless of the execution speeds or failures of the threads.

A lock-free concurrent object implementation is *wait-free* if every thread can complete any operation on the object in a finite number of steps, regardless of the execution speeds or failures of other threads.

A concurrent object implementation is *wait-free with probability* 1 if each thread executing an operation can complete it in a finite number of steps with probability 1, regardless of the execution speeds or failures of the threads.

In this thesis, we implement a shared object that is wait-free with probability 1.

Chapter 4

The CAFÉ Algorithm

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 of bounded size. Each TreeContainer is linearizable with respect to the abortable pool sequential specification introduced in Section 4.1. The algorithm for a single TreeContainer is given in Section 4.2. 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 4.3.

4.1 Abortable Pool Sequential Specification

In this section we introduce a sequential specification called *abortable pool*, which we later implement by the TreeContainer concurrent data structure. An abortable pool supports $put(t) \rightarrow result$ and $get() \rightarrow t$ operations, where result is true or false and t is a task. If put returns true, we say that it *succeeds*, else we say that it *fails*. We assume that tasks inserted in the abortable pool are unique, i.e. if $put(t)$ and $put(t')$ are two different invocations on the container, then $t \neq t'$.

Abortable pool's get operations satisfy the following property: get returns a task t previously inserted by a successful $put(t)$ operation and not returned by any preceding get operation; if there is no such task, it returns $\perp$.

4.2 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.

The code of 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 at 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.

4.2.1 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.

Finding an unoccupied node. The 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 Section 8.3). However, it also creates denser trees with a more balanced node occupation, as we show in Section 8.1.

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

Algorithm 1 TreeContainer, 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 ∨ node.meta[1].p)
11:     then return 1
12:   else return (node.task ≠ ⊥) ? 1 : 0

13: Function put(task):
14:   node ← findNodeForPut(task)
15:   if (node = ⊥) then return false
16:   updateNodeMetadata(node, 1)
17:   return true

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

26: Function putInNode(node, task)
27:   if (node.father ≠ ⊥∧ node.father.task = ⊥)
28:     return putInNode(node.father, task)
29:   if (node.isDirty.CAS(false, true))
30:     node.task ← task; return node
31:   else return ⊥

32: Function get()
33:   while(true):
34:     if (hasTasks(root) = 0) return ⊥
35:     node ← findNodeForGet()
36:     task ← node.task
37:     if (task ≠ ⊥ ∧ node.task.CAS(task, ⊥) = false)
          continue
38:     updateNodeMetadata(node, 0)
39:     if (task ≠ ⊥) return task

40: Function findNodeForGet()
41:   node ← root
42:   while(true)
43:     if(node.task≠⊥ ∨node.meta[0].p=node.meta[1].p=0)
          return node
44:     node ← random child among those with p = 1

45: Function updateNodeMetadata(node, myVal)
46:   trials ← 0;
47:   while(node.father ≠ ⊥)
        ▷ check if my operation has been eliminated
48:     if (myVal ≠ hasTasks(node)) return
49:     if (needToUpdate(node))
50:       trials ← trials +1
51:       if (updateFather(node) ≠ success ∨ trials < 2)
            continue ▷ try again on this node
52:     node ← node.father; trials ← 0

53: Function needToUpdate(node)
54:   fm_1 ← father.meta[node.side].p
55:   if (node.pending > 0) return true
56:   fm_2 ← father.meta[node.side].p
57:   if(fm_1 ≠ hasTasks(node) ∨ fm_1 ≠ fm_2) return true
58:   return false

59: Function updateFather(node)
60:   node.pending.FetchAndInc()
61:   new ← old ← father.meta[node.side]
62:   new.ver ← new.ver +1; new.p ← hasTasks(node)
63:   success ← father.meta[node.side].CAS(old, new)
64:   node.pending.FetchAndDec()
65:   return success
```

at the last level is defined by a parameter k, with higher k's resulting in denser trees. In Section 8.2, 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 the ancestors' metadata. After a task is inserted in node x, the function *updateNodeMetadata()* updates the presence bits of x's ancestors (lines 47–52). 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 implement **updateNodeMetadata** by a direct call to **updateFather**: first read the old presence bit of x's father (line 61), then calculate whether x's subtree contains tasks (line 62), and finally CAS a new metadata value if the old value is incorrect (line 63). 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 that this value is correct, and terminates. Now t_c resumes and successfully changes $x.meta[1].p$ to 0. This makes future gets think that 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 60); after the update, it decrements the counter (line 64). Now, every updating thread first calls the **needToUpdate** function to check the father node's consistency and whether pending updates exist. If the father's state is consistent and there are no pending updates, then the thread can continue without updating the father node. However, if the thread sees $x.pending > 0$ at line 55, it will call **updateFather** (line 51) for x, regardless of x's father's current state. To verify that the father's state has not changed after the the read in line 55, we perform a second read after the check (line 56). Now, in the aforementioned race scenario, t_p sees t_c's pending update and therefore updates the father node using CAS on $y.meta[1].p$, causing failure of t_c's subsequent CAS.

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 7, 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 [12] 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 that y's x-side metadata is now 0, it means there has been a 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 48). This optimization improves performance in scenarios where multiple producers and consumers are working on the same tree.

In Section 9.1, 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 8 that our insertions create a balanced tree, whp. Hence, when the tree contains N tasks, the complexity of a put is $O(\log^2 N)$.

4.2.2 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.

The 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 in the same way as after a task insertion, using *updateNodeMetadata()*.

In Section 9.2, 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.

4.3 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 4.1). The order among the trees in the list is preserved when tasks are returned. 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.

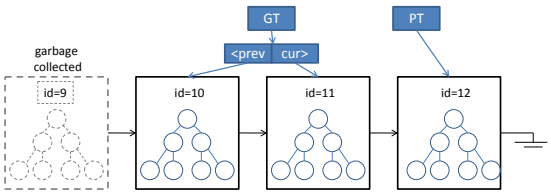


Figure 4.1: CAFÉ keeps a linked list of scalable task trees. The tree height defines the fairness of the protocol.

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 [4]) 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.

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:      ⟨prev, cur⟩: GT                           ▷ tree for consumers
8:      int: oldProducers                         ▷ for producers that move GT
        backwards
9:  Function put(task)
10:     while(true)
11:         latest ← 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 = ⊥) insertNewTree()
17:             PT.CAS(latest, latest.next)
18: insertNewTree()
19:     newNode ← Node()
20:     cur ← PT ▷ go to the end of the list
21:     for(; cur.next ≠ ⊥; cur ← cur.next);
22:     newNode.id ← cur.id +1
23:     cur.next.CAS(⊥, newNode) ▷ return even if CAS fails
24: Function moveGTBack(Node: prodTree)
25:     oldProducers.FetchAndInc()
26:     while(true)
27:         gtVal ← GT
28:         if (gtVal.cur.id ≤ latest.id) break
29:         newGT ← ⟨⊥, latest⟩
30:         if (GT.CAS(gtVal, newGT) = true) break
31:     oldProducers.FetchAndDec()
32: Function get()
33:     ptVal ← PT
34:     gtVal ← GT
35:     while(true)
36:         task ← gtVal.prev.getTask()
            if (task ≠ ⊥) return task
37:         task ← gtVal.cur.getTask()
            if (task ≠ ⊥) return task
            ▷ could not find a task in the tree
38:         if (ptVal.id ≤ gtVal.cur.id) return ⊥
39:         if (oldProducers = 0) then
40:             newGT ← ⟨gtVal.cur, gtVal.cur.next⟩
41:             GT.CAS(gtVal, newGT)
42:             gtVal ← GT
43:         else
44:             gtVal ← ⟨gtVal.cur, gtVal.cur.next⟩
```

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 $\langle \perp, T \rangle$ in the function $moveGTBack$.

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.

Chapter 5

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 Chapters 7– 9. Recall that we assume that an adversary controls thread scheduling but cannot influence the randomness threads use. In Section 5.1, we state key lemmas for CAFÉ's linearizability proof. Section 5.2 states key liveness lemmas. Finally, Section 5.3 states key lemmas of TreeContainer's performance properties. Together, the lemmas in Sections 5.1 and 5.2 imply the following theorem:

Theorem 1. *CAFÉ implements a linearizable α-fair task pool that is wait free with probability 1.*

5.1 Safety Properties

We start by showing that CAFÉ implements a linearizable task pool. Intuitively, if the task pool is nonempty, then a get must be able to find a task. Formally, we prove in Lemma 10 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 2 that each TreeContainer that CAFÉ uses is itself linearizable with respect to the abortable task pool specification. Second, we show in Theorem 3 that CAFÉ with a TreeContainer of size α is linearizable with respect to α-fair task pool specification.

The key to proving Theorem 2 is Lemma 7, 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 task as long they have not all been removed. We briefly describe the proof of Lemma 7. Let x and y be two nodes that 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 Theorem 3. The proof of Theorem 3 consists of the get validity property provided by Lemma 9 and a proof of α-fairness. Lemma 9 shows that after a put inserts a task in some TreeContainer, subsequent get operations will not skip this TreeContainer when looking for a task. 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. Because 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 that $GT.cur$ 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.

To prove α-fairness we show that the gets return tasks preserving order among the TreeContainers they were inserted into. This follows simply because tasks are inserted and removed based on the linked list order of the TreeContainers.

5.2 Liveness Properties

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 determinis-

tically 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 lemmas, proven in Chapter 9.

Lemma 1. *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 1, we prove the following. Note that CAFÉ using TreeContainers of height 0 is equivalent to a linked list.

Lemma 2. *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 Section 9.2.

Lemma 3. *Every get operation of CAFÉ terminates in a finite number of steps.*

5.3 Performance Properties

We first show that our trees are dense: by choosing an appropriate k (number of trials at last level of TreeContainer) 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 Section 8.1, 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 Section 8.2:

Lemma 4. *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 Section 8.3):

Lemma 5. *Consider a TreeContainer after N successful* put *operations. Then each of these operations has taken $O(\log^2 N)$ steps with probability at least $1 - \frac{1}{2(N+1)^{\frac{4}{3}}}$.*

Chapter 6

Evaluation

In this chapter we evaluate the performance of our Java implementation of CAFÉ and compare it to other algorithms in Section 6.1. In Section 6.2 we analyze the performance of the algorithms. Section 6.3 considers the influence of tree height on CAFÉ. Section 6.4 investigates the reasons for CAFÉ's superior performance. Finally, Section 6.5 considers the cost of implementing the fairness requirements.

6.1 Experiment Setup

We compare the following task pool implementations:

- **CAFÉ-h** – CAFÉ with height h for each tree. Unless stated otherwise, we use $h = 13$.
- **CLQ** – The standard Java 6 implementation of a (FIFO) lock-free queue by Michael and Scott [11] (class `java.util.concurrent.ConcurrentLinkedQueue`), which is considered to be one of the most efficient lock-free algorithms in the literature [8, 10].
- **LBQ** – The standard Java 6 implementation of a (FIFO) lock-based 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.

- **SQ** – The original segmented queue implementation of [2] (downloaded from the web page of the project). SQ is a non-FIFO queue. Like CAFÉ, SQ provides a bound on the number of overtaken tasks and maintains a linked list of segments. Each segment is an array of nodes with a size of the fairness bound. Unless stated otherwise, we use SQ with segment size of 255, which we found to provide maximal throughput.

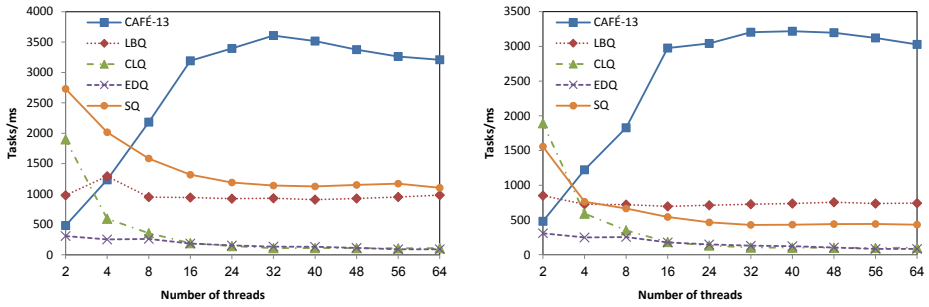
We use a synthetic benchmark for the performance evaluation, in which producer threads work in loops inserting dummy items, and consumer threads work in loops retrieving dummy items.

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 AggressiveHeap flag on. We run up to 64 threads on the 32 cores. The influence of garbage collection was negligible for all algorithms[1].

6.2 System Throughput

Workloads with the same number of producers and consumers. In Figure 6.1 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 lock-based queue. Producers of CAFÉ are almost three times faster than those of SegmentedQueue, and consumers more than six times faster. When the number of working threads exceeds the number of hardware

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



(a) Task insertion rate.

(b) Task removal rate.

Figure 6.1: 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.

threads in the system, the throughput of CAFÉ decreases moderately, but still outperforms the other algorithms.

As we can see in Figure 6.1, the results of both Michael-Scott concurrent queue and ED pools are worse 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 multiprocessor systems: accessing a memory location from multiple processors is significantly slower than accessing it from multiple hardware threads on the same chip, which usu-

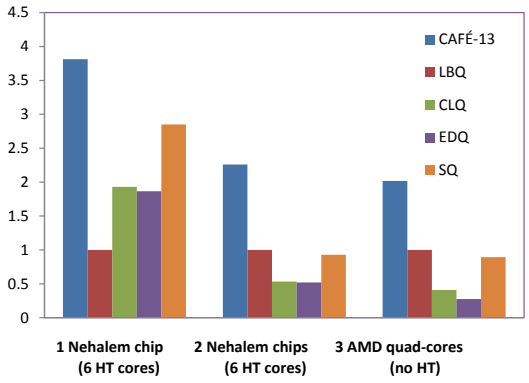


Figure 6.2: Throughput on different hardware architectures, normalized by the throughput of LBQ. There are 6 producer threads and 6 consumer threads.

ally share a last-level cache. We now show how the non-uniformity of memory accesses influences performance.

Figure 6.2 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 lock-free 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. 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 access to a machine with more than 12 HW threads per chip.

Workloads with a high number of producers and a varying number of 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 are typically part of a critical path. The insertion throughput should remain high even if there are more producers than consumers. To model this behavior, we run algorithms with more producers than consumers.

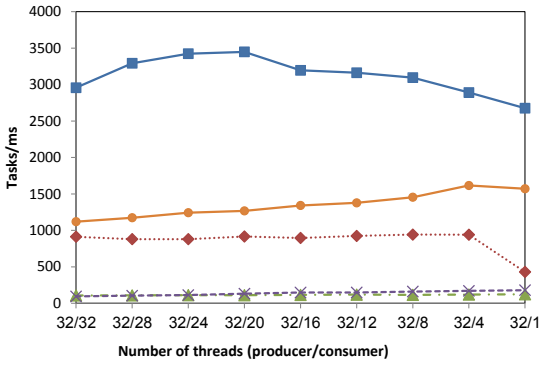
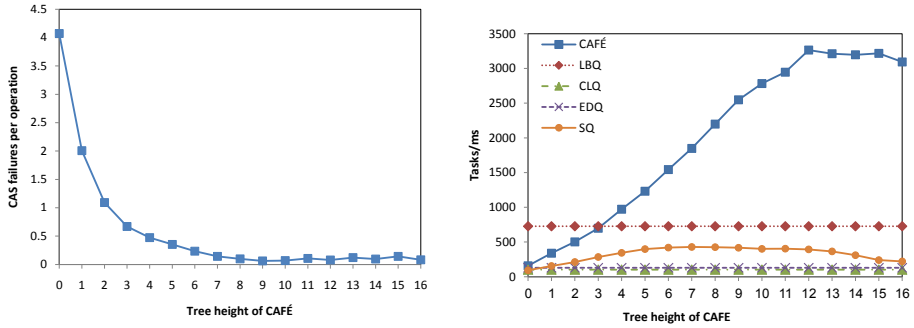


Figure 6.3: Task insertion rate for big number of producers, various number of consumers. Insertion rate of CAFÉ remains significantly higher than that of the competitors.

We demonstrate the task insertion rate of the algorithms for the workloads with different numbers of consumers in Figure 6.3. The insertion rate of CAFÉ remains high for all configurations. SegmentedQueue pro-

ducers show the second best results. SQ's throughput continues to grow as the number of producers becomes higher. However, CAFÉ is still two times faster thanks to the scalability of TreeContainer.

6.3 Choosing the Tree Height



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

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

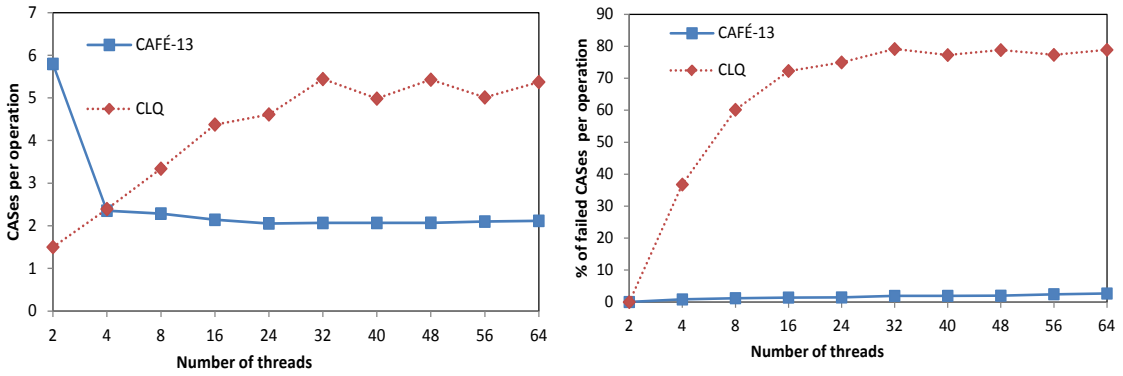
Figure 6.4: CAS failures and system throughput as a function of CAFÉ's tree height for 16 producers and 16 consumers. Small trees induce high contention because of linked list manipulations and reduced tree randomization. Excessively large trees induce contention among producers and consumers operating in the same tree. The SQ pool throughput on 6.4(b) is measured for segments of sizes equal to TreeContainer of the specified height.

In Figure 6.4 we demonstrate CAFÉ's performance for 16 producers and 16 consumers as a function of tree height. Figure 6.4(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 6.4(b). Increasing the tree height improves throughput up to a certain point (height 13 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 14, most of the threads operate in the same tree, which increases contention and decreases performance. We can see in Figure 6.4(b) that using SegmentedQueue with segment sizes equal to CAFÉ's TreeCon-

tainers does not show significant throughput improvement, i.e. for segments sizes bigger than 255, the further FIFO relaxation of SegmentedQueue does not bring any benefit.

6.4 Performance Breakdown



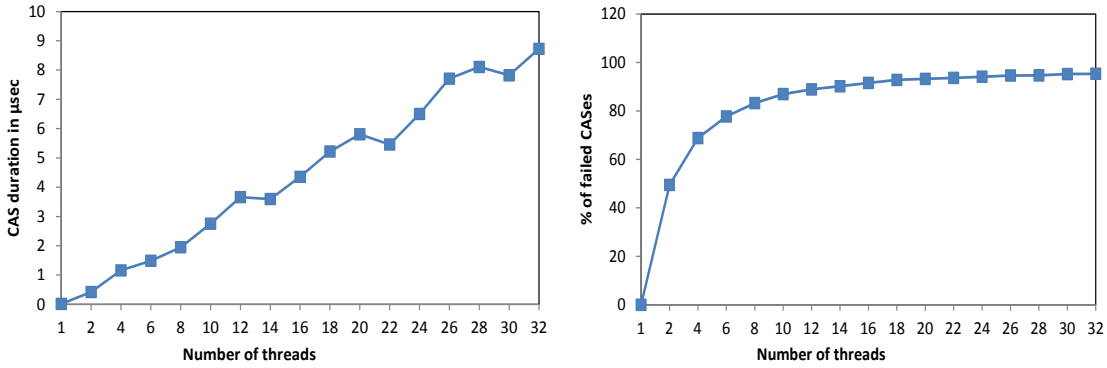
(a) CAS instructions per operation.

(b) Percent of failed CAS instructions.

Figure 6.5: CAFÉ and CLQ CAS instruction statistics for producer threads.

We saw in previous experiments that CAFÉ outperforms all competitors. To better understand the reasons for these results, we break down the performance costs of CAFÉ and CLQ. There are a number of factors contributing to CAFÉ's superior performance. First, we notice that CAFÉ performs fewer CAS instructions per operation. In Chapter 8 we formally prove that under *worst case* scheduling, producer threads of CAFÉ perform asymptotically fewer CAS instructions than those of CLQ. We now observe this effect also in practice: we measure CAS instructions per operation. In Figure 6.5(a) we present the number of CAS instructions per operation for producer threads. We see that CLQ performs up to 3 times more CAS instructions per operation than CAFÉ. Note that CLQ mostly runs CAS instructions, while CAFÉ also has an additional overhead for maintaining complex data structures. Therefore, if this were the only factor contributing to the performance gap, we could expect CAFÉ's throughput to be less than 3 times higher than that of MSQ. However, CAFÉ outperforms CLQ by as much as a factor of 30. The main reason for this is that CAFÉ's CAS loops are less expensive because fewer of them contend on the same shared resource. We can see the degree of contention in Figure 6.5(b).

Most CAS instructions of CLQ fail, while CAFÉ's CASes seldom fail even for high numbers of threads.



(a) Duration of CAS instruction (μsec).

(b) Percent of failed CAS instructions.

Figure 6.6: CAS instructions cost depending on number of threads competing on the same shared resource. Increased contention on the shared object increases the duration of CAS instructions.

To illustrate the impact of contention on the cost of CAS instructions we conduct another experiment. The experiment runs a predefined number of threads that operate on a shared variable of *AtomicReference* type. Every thread runs two operations in a loop for a predefined period of time: it reads a shared atomic reference value and then tries to update the shared variable to a new value using a CAS instruction. The average duration of a CAS instruction is presented in Figure 6.6(a) and the percent of failed CASes is shown in Figure 6.6(b). We can see from Figure 6.6(b) that increasing the number of threads concurrently CASing the same shared object increases the percent of CAS failures, while Figure 6.6(a) shows that in experiments with a high percent of failures, the duration of CAS operations grows by up to 18 times. To get a feeling for how much faster CASes of CAFÉ can be compared to those of CLQ, we use the percentage of CAS failures as a measurement for contention. For example, from Figure 6.5(b) we see that with 32 threads, CLQ has 80% CAS failures. We get this contention level with 8 threads in Figure 6.6(b) where the CAS duration is around 2μs. CAFÉ's percentage of CAS failures is close to zero, which corresponds to a CAS duration of around 0.01 μs. According to these calculations, CAFÉ's CASes are roughly 20 times faster than those of CLQ.

6.5 The Cost of Fairness

When we are not concerned with fairness requirements we can construct simple and scalable task pools from existing containers using a partitioning technique, as we now describe: In this section we introduce such non-fair task pools and measure their performance. We show that fairness has a high price. Once we forfeit fairness requirements we can simply partition existing task pools and get more than 1.5 times higher throughput than that of CAFÉ, which is, according to Section 6.2, the best among existing fair containers. We show that the partitioning technique, by itself, can bring major throughput improvements, e.g., a factor of 10 for the CLQ task pool. We further show that a correct NUMA-aware assignment of threads to cores can yield significant additional improvements, more than a factor of 3 for some pools.

6.5.1 Partitioning Technique

We now describe the partitioning technique used for the construction of non-fair task pools. Given a basic task pool data structure, a number of producer and consumer threads, and a number of partition groups, we implement partitioning as follows:

1. Divide threads to the desired number of groups, s.t. every group has at least one producer and consumer thread.

2. Allocate to each group a basic task pool.

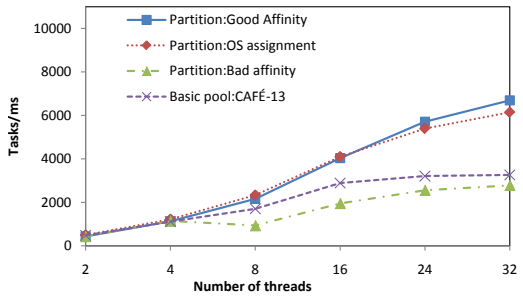
3. Each producer thread inserts tasks into its group's pool.

4. Each consumer thread:

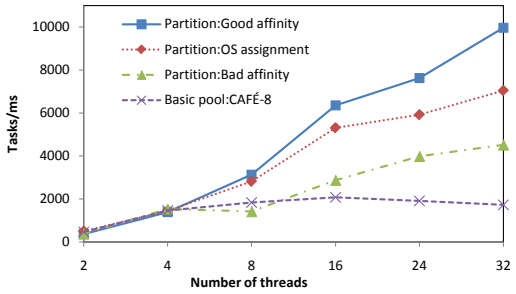
   (a) Tries to retrieve a task from the pool of its group.

   (b) If the group's pool is empty, it traverses in random order one by one other groups' pools and tries to retrieve a task from them.

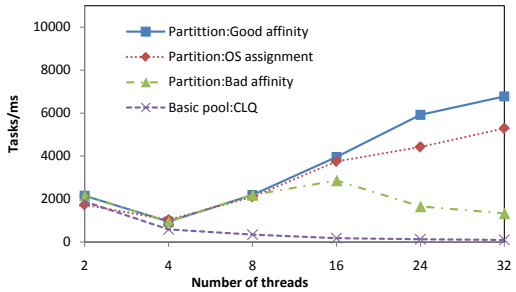
   (c) If all the pools are empty, it returns $\perp$.



(a) Partitioning of CAFÉ-13.



(b) Partitioning of CAFÉ-8.



(c) Partitioning of CLQ pool.

Figure 6.7: Here we show throughput of containers constructed using a partitioning technique for three different basic pools: CAFÉ -13 (6.7(a)), CAFÉ -8 (6.7(b)) and CLQ (6.7(c)). Evaluation is done with equal numbers of producer and consumer threads. Groups of size $\lceil \frac{\#threads}{4} \rceil$ are used for partitioning due to the maximal throughput achieved by such configuration. Each plot presents results for three types of thread assignment: *Good affinity* - threads in the same groups are assigned to cores of a single chip; *OS assignment* – threads are dynamically assigned by the scheduler of Linux OS with 2.6.35 kernel version; *Bad affinity* –threads in the same group are assigned to cores of different chips. Basic pools use OS assignment for threads.

6.5.2 Throughput of Partitioned Task Pools

We run the synthetic benchmark used for the evaluation in previous sections for partitioned task pools based on CLQ and CAFÉ task pools. The evaluation results are presented in Figure 6.7. The graphs show results for three types of thread assignment: *good affinity* - threads are optimally assigned to cores to preserve locality of partitioned pools; *OS assignment* – threads are dynamically assigned by the scheduler of Linux OS with 2.6.35 kernel version; *Bad affinity* –threads are intentionally assigned to break locality. Basic pools use OS assignment for threads.

As we saw in Section 6.2, CAFÉ outperforms existing state-of-the-art task pools that

have fairness guarantees. However, as Figure 6.7 shows, once we omit the fairness requirement, by non-fair partitioning of CLQ pool under OS scheduling (Figure 6.7(c)), we get throughput more than 1.5 times higher than that of CAFÉ.

We see in Figure 6.7 that for basic pools that do not have high cache utilization (Figures 6.7(b) and 6.7(c)) partitioning, even with a bad thread assignment, improves throughput from 2 to 10 times. The exception is CAFÉ-13 (Figure 6.7(a)), where every group allocates relatively big TreeContainer of size $2^{14} - 1$ nodes. Hence, when multiple containers are allocated on a single chip and concurrently used by threads of multiple groups, cache efficiency decreases. As a result, partitioning of CAFÉ -13 does not bring any benefit. However, as we see in Figure 6.7(b), if we use smaller TreeContainers in CAFÉ, we get more than a factor of 2 improvement over the basic pool for partitioning with bad affinity and get the highest throughput.

For all containers we get maximal throughput when threads in partitioned groups are assigned to cores of a single chip, i.e., have a good affinity. We see in Figure 6.7, that by only changing the threads assignment we can increase the system throughput by up to factor of 4 for the CLQ pool and more than twice for CAFÉ-8.

Chapter 7

Proofs of Safety Properties

In this section, we first show that TreeContainer is linearizable with respect to the abortable pool sequential specification, defined in Section 4.1. Then, basing on TreeContainer linearizability, we prove that CAFÉ using TreeContainer of size α is linearizable with respect to α-fair task pool sequential specification, defined in Section 3.3 . We will refer to line number r in Algorithms 1 and 2 by $\langle Tr \rangle$ and $\langle Cr \rangle$, respectively.

7.1 Safety of TreeContainer

We now show that a single TreeContainer is linearizable with respect to abortable pool specification defined in Section 4.1. To prove the linearizability we need to prove that a get operation only returns $\perp$ if the tasks of all successful put operations that completed before the get have been taken. The basic idea, stated in Lemma 7, is that between the time a successful 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 that 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 6. *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 $\langle T63 \rangle$) $C_1, \ldots, C_m$ on $y.meta[x.side]$, and*

let $R_1, \ldots, R_m$ be the corresponding reads (at $\langle T61 \rangle$) preceding each such CAS. Then for $i = 2, \ldots, m$, R_i occurs after C_{i-1}.

Proof. From $\langle T62 \rangle$ and $\langle T63 \rangle$, 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 refer to the value that the function **hasTasks**(x) would return if evaluated at time t.

Lemma 7. *Consider any TreeContainer T, and let p be a put operation that inserted a task in node $x_0 \in T$ and completed before time τ. Suppose further that by 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 1. *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' such that $t < t' < \tau$, and $p.node = y$ at t', and $hasTasks(y) = 1$ from t' until τ.*

Proof. Since p finishes its operation by time τ, then it returns either after not entering the loop in $\langle T47 \rangle$ or at $\langle T48 \rangle$. 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 does not return from $\langle T48 \rangle$ while $p.node = x$. Thus, p performs $\langle T52 \rangle$ at some time $t' > t$, and the first part of the claim holds.

To show the second part, consider 3 cases for the iteration when $p.node = x$: (1) the if on $\langle T49 \rangle$ is false at some point, (2) the if on $\langle T49 \rangle$ is true and the if on $\langle T51 \rangle$ is true due to at least one successful **updateFather** call, or (3) the if on $\langle T49 \rangle$ is true and **updateFather** on $\langle T51 \rangle$ is false twice.

In case (1), p does not call **updateFather**. We will show that in this case $hasTasks(y)$ is true to begin with and any other thread that updates y's metadata keeps the correct value of $hasTasks(y)$. Denote t'' the time when the if on $\langle T49 \rangle$ samples false. For a line number i, denote by t_i^p the line at which p executes line i in the call **needToUpdate** that completes at t''. We get, $t_{54}^p < t_{55}^p < t_{56}^p < t_{57}^p < t''$. Note that in line 54 $hasTasks(y) = 1$. We next show that this persists until τ. Consider any step that changes the value of $y.meta[x.side].h$ between t_{54}^p and τ. By inspection, we see that this must be a successful CAS step on $\langle T63 \rangle$, say by thread q, occurring at time t_{63}^q and denote t_{61}^q the time of q's preceding read. Note that q increments $x.pending$ before t_{61}^q and decrement it after t_{63}^q. Let t_{60}^q and t_{64}^q be the times of these events respectively. We observe that the mentioned events of q have the following order: $t_{60}^q < t_{61}^q < t_{63}^q < t_{64}^q$. Recall that $\langle T49 \rangle$ samples false, therefore the if on line 55 is false, and, either $t_{55}^p < t_{60}^q$ or $t_{55}^p > t_{64}^q$. We now show that in both cases $hasTasks(y) = 1$ from t_{55}^p to τ. Assume first $t_{55}^p < t_{60}^q$, then $t_{54}^p < t_{60}^q$, thus, $t_{54}^p < t_{61}^q$ and so $t < t_{61}^q$. Recall that from t till τ $hasTasks(x) = 1$, so in line 61, q reads 1 and writes 1 to $y.meta[x.side].h$, which keeps $hasTasks(y) = 1$. The second case is handled by the second read of the father's metadata in line 56: $t_{55}^p > t_{64}^q$, therefore $t_{56}^p > t_{63}^q$. $\langle T49 \rangle$ is false, so at moment t_{56}^p p reads value 1, therefore, again, in line 63 q keeps value 1 of $y.meta[x.side].h$.

In case (2), let t'' be the time when p did a successful CAS to $y.meta[x.side]$ in $\langle T63 \rangle$ 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 on $\langle T61 \rangle$ by q right before C. Then R occurs after t'', by Lemma 6. So R reads $hasTasks(x) = 1$ sometime after $t'' > t$, and after C, we have $y.meta[x.side].h = 1$.

In case (3), let C_1, C_2 be the two failed CASes on $\langle T63 \rangle$ that p performed during its 2 executions of **updateFather**, and let R_1, R_2 be the reads on $\langle T61 \rangle$ 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 $\langle T61 \rangle$ by q before C'_2. Then by Lemma 6, 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 $\langle T62 \rangle$ after R'_2. Thus, $y.meta[x.side].h = 1$ after C'_2. □

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 1, 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 that 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 by way of 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 7, 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 2. *TreeContainer implements a linearizable producer-consumer pool with respect to the abortable task pool sequential specification.*

Proof. By using a CAS in $\langle T37 \rangle$, we ensure that every task can be returned by at most one get. 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. □

7.2 Safety of CAFÉ

We now show that CAFÉ implements a linearizable task pool with respect to α-fair task pool sequential specification.

The proof of CAFÉ get validity property is based on the fact that CAFÉ uses a list of linearizable TreeContainers. We show in Lemma 9 that if there is an unremoved task in some TreeContainer T, then a get operation will start looking for tasks starting from T or an earlier TreeContainer. This implies that if a get operation returns $\perp$, then all the tasks that were put into some TreeContainer before the get started have been removed. This implies get validity of CAFÉ operations.

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 8. *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 9. *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 performed $\langle C27 \rangle$, and in the second case, let t be when p performed $\langle C30 \rangle$. The lemma follows from the following claim.

Claim 2. *$GT.cur.id \leq T.id + 1$ 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 8, 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 8. 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 10. *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 9, 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. □

The following lemma and corollary prove the α-fairness property of CAFÉ pool.

Lemma 11. *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 9, 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 performs $T_1.getTask()$ 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. □

Corollary 2. *For a TreeContainer of size $\alpha = 2^{h+1} - 1$, CAFÉ is α-fair.*

Proof. In order to succeed, TreeContainer put operation first has to win a CAS on some node at $\langle T29\rangle$. The first winning put permanently changes the node state to dirty and, hence, all following CASes on this node at $\langle T29\rangle$ will fail. Therefore, the number of successful put operation on a single TreeContainer is bounded by number of nodes in TreeContainer, i.e. by α. Therefore, if between termination of p_1 put operation and invocation of p_2 put operation α other successful puts have occurred, the task of p_1 and the task of p_2 reside in different TCs. Let T_1 and T_2 denote the TCs, respectively. Then, because p_1 terminates before invocation of p_2, $T_1.id < T_2.id$. Hence, by Lemma 11, get operations that start after termination of p_2 will retrieve p_1's task before the task of p_2, and the corollary follows. □

Theorem 3. *CAFÉ with TreeContainer of size α is linearizable with respect to α-fair task pool specification.*

Proof. The theorem follows from Lemma 10 and Corollary 2. □

Chapter 8

Proofs of Performance Properties

We now formally prove the algorithm properties that were presented in Section 5. 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.

8.1 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 3. *If tasks are inserted into TreeContainer by RW, the probability that an insertion*

increases a current value of h_t *is*

$$Pr_{RW}(\textit{increase } h_t) = \frac{X_{h_t}}{2^{h_t}} \tag{8.1}$$

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 4. *Assume that* N *tasks have been inserted into TreeContainer by RW. Then the probability that the next insertion by RW increases* h_t *is bounded by*

$$Pr_{RW}(\textit{increase } h_t) \leq \frac{N+1}{2^{h_t+1}} \tag{8.2}$$

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 3, we get (8.2). □

Claim 5. *The probability to increase* h_t *by inserting a task with a TreeContainer* put() *operation is*

$$Pr(\textit{increase } h_t) = (Pr(\textit{increase } h_t \textit{ by RW}))^{r_{h_t}+1} \cdot Pr(\textit{reached level } h_t) \tag{8.3}$$

Proof. The TreeContainer *put()* function calls *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

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

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 3, $Pr(\textit{increase } h_t \textit{ by RW}) = \frac{X_{h_t}}{2^{h_t}}$, so we get (8.3). □

Lemma 12. *Insertion by a 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 5. □

8.2 TreeContainer Density Guaranties

Lemma 13. *Assume that N tasks that have been inserted into an empty TreeContainer using a* put() *function. Then the probability that the last occupied level h_t does not exceed h_u, $0 < h_u \leq h$, 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

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

When insertion i samples tree nodes, there are at most $i-1$ occupied nodes in the tree. By Claims 4, 5

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

Thus,

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

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 4 (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 the last level of TreeContainer there are k trials to occupy node in *findNodeForPut()* procedure, i.e. $r_h = k$. By Lemma 13, 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 $1 - \frac{2^{\frac{(k+2)^2}{k+3}\cdot h}}{2^{(h+1)\cdot(k+1)}} = 1 - \frac{1}{2^{(3-\frac{7}{k+3})\cdot h+k+1}}$. □

8.3 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 a CAS operation.

Claim 6. *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 statement of the lemma, the last occupied level is h_t, and so *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). The 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 the *isDirty* variable of the nodes along the way. If some CAS succeeds, the reservation has succeeded. 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 7. *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 the *trials* variable in lines 50 and 51 of TreeContainer). Hence, the total number of steps is $O(h_v)$. □

Lemma 14. *Every* put() *operation of TreeContainer makes at most $O(h^2)$ steps.*

Proof. The function *put()* of TreeContainer performs one call to *findNodeForPut()* and at most one call to *updateNodeMetada()* functions. The lemma follows from Claims 6, 7. □

Lemma 5 (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 4, 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 4 the last occupied level is at most $\frac{4}{3} \cdot log_2(N)$ with probability at least $1 - \frac{1}{2 \cdot (N+1)^{\frac{4}{3}}}$.

The 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 6, 7 each of these operations makes at most $O(log^2(N))$ steps. □

Chapter 9

Proofs of Liveness Properties

In this section we provide a formal proof that consumers operations are wait-free and producers operations are wait-free with probability 1.

9.1 Probabilistic Wait Freedom of Producers

Lemma 1 (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 $1 \leq i < 2^h$, $Pr(S_T|E_i) = 1 - Pr(\neg S_T|E_i) = 1 - (\frac{i}{2^h})^k$; for $i = 2^h$, $Pr(S_T|E_{2^h}) = 0$ (all nodes of C are occupied). Thus,

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

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

$$Pr(S_T) \geq [1 - (\frac{2^h - 1}{2^h})^k] \cdot (1 - Pr(E_{2^h})) \tag{9.1}$$

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

$$\begin{aligned} Pr(E_{2^h}) = & \sum_{j=1}^{P-2} [(1 - (1 - \tfrac{1}{2^h})^{k\cdot(P-j)}) \cdot Pr(A_j)] \\ & \leq [1 - (1 - \tfrac{1}{2^h})^{k\cdot(P-1)}] \cdot \sum_{j=0}^{t-2} Pr(A_j) \leq 1 - (1 - \tfrac{1}{2^h})^{k\cdot(P-1)} \end{aligned}$$

By substituting the inequality result into (9.1), we finish the proof. □

Lemma 15. *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 $GT.curr.id \leq id_{t_1}$. 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 $id \geq id_{t1}$. 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 2 (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 14, every TreeContainer's put operation takes $O(h^2)$ steps. By Lemma 1, 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 15, the second stage terminates in a finite number of steps, which finishes the proof. □

9.2 Consumers Wait Freedom

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

Lemma 16. *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, in 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 7, updateNodeMetada() makes at most $O(h)$ steps. Hence, the total number of steps during the *get()* operation is $O(h \cdot 2^h)$. □

Lemma 17. *Every* get() *operation of TreeContainer terminates within 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 14, 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 16, 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 3 (restated). Every get operation of CAFÉ terminates within a finite number of steps.

Proof. Let T_c be a consumer thread performing a *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 17, 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()* operations 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 the 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. □

We conclude with the following theorem.

Theorem 1 (restated). *CAFÉ implements a linearizable α-fair task pool that is wait free with probability 1.*

Chapter 10

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 experiments 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 that 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?

Bibliography

[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] Y. Afek, G. Korland, and E. Yanovsky. quasi-linearizability: relaxed consistency for improved concurrency. In *Proceedings of the 14th international conference on Principles of distributed systems*, OPODIS'10, pages 395–410, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. CAFÉ: Scalable task pools with adjustable fairness and contention. In D. Peleg, editor, *DISC*, volume 6950 of *Lecture Notes in Computer Science*, pages 475–488. Springer, 2011.

[4] 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.

[5] 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.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium*

on computer architecture, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

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

[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

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

[11] 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.

[12] 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.

[13] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.