SnapQueue: Lock-Free Queue with Constant Time Snapshots

Aleksandar Prokopec

École Polytechnique Fédérale de Lausanne, Switzerland
aleksandar.prokopec@alumni.epfl.ch

Abstract

We introduce SnapQueues - concurrent, lock-free queues with a linearizable, lock-free global-state *transition* operation. This transition operation can atomically switch between arbitrary SnapQueue states, and is used by enqueue, dequeue, snapshot and concatenation operations. We show that implementing these operations efficiently depends on the persistent data structure at the core of the SnapQueue.

This immutable *support* data structure is an interchangeable kernel of the SnapQueue, and drives its performance characteristics. The design allows reasoning about concurrent operation running time in a functional way, absent from concurrency considerations. We present a support data structure that enables $O(1)$ queue operations, $O(1)$ snapshot and $O(\log n)$ atomic concurrent concatenation. We show that the SnapQueue enqueue operation achieves up to 25% higher performance, while the dequeue operation has performance identical to standard lock-free concurrent queues.

Categories and Subject Descriptors E.1 [*Data Structures*]: Lists, stacks and queues

Keywords queues, lock-free, concatenation, snapshots

1. Introduction

Scalability in a concurrent data structure is achieved by allowing concurrent accesses to execute independently on separate parts of the data structure. While efficient concurrent implementations exist for many different data structure types [15], efficient operations that change their global state are still largely unexplored. For example, most concurrent hash tables [7] [13], concurrent skip lists [21], and concurrent queues [14] [22] do not have an atomic snapshot operation, size retrieval or a clear operation.

SCALA'15, June 13, 2015, Portland, OR, USA
Copyright 2015 ACM 978-1-4503-3626-0/15/06...$15.00
http://dx.doi.org/10.1145/2774975.2774976

Concurrent queues are used as buffers between producers and consumers in streaming platforms, as event queues in reactive programming frameworks, or mailbox implementations in actor systems. Traditionally, concurrent queues expose enqueue and dequeue operations. Augmenting them with atomic global operations opens several new use cases:

- *Persisting actor state:* Actor frameworks expose persistence modules to persist and recover actor state [1]. Persisting the state requires blocking the mailbox until the pending messages are copied. With an efficient snapshot operation, the mailbox can be persisted in real-time.
- *Forking actors:* Adding a `fork` or a `choice` operator to the actor model [26] requires copying the contents of the mailbox. To achieve atomicity, the corresponding actor is blocked during the copying and cannot access the mailbox. Efficient snapshots help avoid this problem.
- *Dynamic stream fusion:* Streaming platforms express computations as dataflow graphs. Each node in this graph executes a modular subset of the computation, but also introduces some buffering overhead [24]. Runtime optimizations eliminate this overhead by fusing subsets of the graph. Efficient, atomic buffer concatenation and joining allow optimizations without blocking the computation.

This paper introduces a lock-free queue implementation, called SnapQueue, which, in addition to enqueue and dequeue operations, exposes an efficient, atomic, lock-free transition operation. The transition operation is used to implement snapshots, concatenation, rebalancing and queue expansion. Although SnapQueue is a concurrent data structure, it relies on a persistent data structure to encode its state. We describe the SnapQueue data structure incrementally:

1. We describe *lock-free single-shot queues*, or *segments*, and their enqueue and dequeue operations in Section 2.
2. We show that segments can be atomically frozen, i.e. transformed into a persistent data structure.
3. We describe SnapQueues and their fundamental atomic operation called *transition* in Section 3.
4. We implement SnapQueue enqueue, dequeue, snapshot and concatenation using the transition operation.

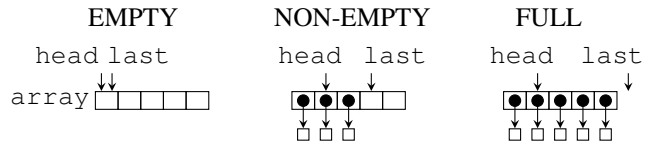

Figure 1. Single-Shot Queue Illustration

```
class Segment[T](
  val array = new VolatileArray[T],
  @volatile var head: Int = 0,
  @volatile var last: Int = 0)
```

Figure 2. Single-Shot Queue Data Type

5. We show how to tune the running time of SnapQueue operations using persistent data structures in Section 4.
6. We evaluate SnapQueues against similar concurrent queue implementations in Section 5, and show that having snapshot support adds little or no overhead.

The goal of the paper is not only to propose a novel concurrent, lock-free queue with atomic constant time snapshots, but also to reproach the common belief that *persistent data structures are slow, and consequently irrelevant for high-performance parallel and concurrent computing*. As this paper shows, persistent data structures can simplify the development of and reasoning about concurrent data structures.

For the sake of conciseness, code listings slightly diverge from valid Scala code in several ways. First, atomic `READ`, `WRITE` and `CAS` operations differ from standard Scala syntax, which relies on the `Unsafe` class. These operations retain the standard Java volatile semantics, and are used to access volatile object fields. Second, volatile arrays are represented with a non-existent `VolatileArray` class. Finally, generically typed methods sometimes accept or return special singleton values. This does not type-check in Scala, but can be worked around at a small syntactic cost. The complete SnapQueue implementation can be found in our online code repository [2].

2. Single-Shot Lock-Free Queue

We start by examining a simplistic lock-free queue implementation, called a *single-shot lock-free queue*. This queue is *bounded* – it can contain only up to L elements. Second, only up to L enqueue and up to L dequeue operations can be invoked on this queue. Despite these limited capabilities, single-shot lock-free queue is the basic building block of the SnapQueue, as we show in Section 3.

Single-shot queue is defined by a single data type called `Segment`, shown in Figure 2. `Segment` contains an `array` of queue elements, initially filled with special `EMPTY` values, `head` – the position of the first element in the queue, and `last` – the estimated position of the first `EMPTY` array entry.

```
1 @tailrec def enq(p: Int, x: T): Boolean =
2   if (p >= 0 && p < array.length) {
3     if (CAS(array(p), EMPTY, x)) {
4       WRITE(last, p + 1)
5       true
6     } else enq(findLast(p), x)
7   } else false
8 @tailrec def findLast(p: Int): Int = {
9   val x = READ(array(p))
10  if (x == EMPTY) p
11  else if (x == FROZEN) array.length
12  else if (p + 1 == array.length) p + 1
13  else findLast(p + 1)
14 }
```

Figure 3. Single-Shot Queue Enqueue Operation

As we will see in Section 2.2, a single-shot queue can become *frozen*, in which case no subsequent enqueue or dequeue operations can succeed. In this frozen state, `array` may contain a special `FROZEN` value.

2.1 Basic Operations

In this section, we study the basic single-shot queue operations. The enqueue operation overwrites an `EMPTY` entry in the `array`. At all times, it ensures that the `array` corresponds to the string $\texttt{REMOVED}^p \cdot \texttt{T}^n \cdot \texttt{EMPTY}^m$, where `T` is the element type, and `array` length is $L = p + n + m$. After inserting an element, enqueue sets the `last` field to point to the first `EMPTY` entry.

We define an auxiliary `enq` method in Figure 3, which takes an estimated position `p` of the first `EMPTY` entry, and an element `x`. The `enq` first checks that `p` is within the bounds of `array`. It then attempts to atomically CAS an `EMPTY` entry with `x` in line 3. An unsuccessful CAS implies that another `enq` call succeeded, so `findLast` finds the first special entry, and `enq` is retried. A successful CAS means that `x` is enqueued, so the value `last` is increased in line 4. Due to potential concurrent stale writes, `last` is an underestimate.

The `enq` precondition is that `p` is less than or equal than the actual first `EMPTY` entry position. This is trivially ensured by specifying the `last` field when calling `enq`.

Lemma 1. *If `enq` returns `true`, then its CAS operation added an element to the queue. Otherwise, the queue is either full or frozen.*

Dequeue atomically increments the `head` field to point to the next element in the queue. Unlike `last`, the `head` field always precisely describes the position of the first unread element. When `head` becomes larger than `array` length, the queue is either empty or frozen. Similarly, when `head` points to `EMPTY` or `FROZEN`, the queue is considered empty or frozen, respectively.

The `deq` method in Figure 4 starts by reading `head` to a local variable `p`, and checks if `p` is within bounds. If it is greater than the `array` length, the queue is empty. If `p` is

```
15 @tailrec def deq(): T = {
16   val p = READ(head)
17   if (p >= 0 && p < array.length) {
18     val x = READ(array(p))
19     if (x == EMPTY || x == FROZEN) NONE
20     else if (CAS(head, p, p + 1)) {
21       WRITE(array(p), REMOVED)
22       x
23     } else deq()
24   } else NONE // used-up or frozen
25 }
```

Figure 4. Single-Shot Queue Dequeue Operation

```
26 def freeze() {
27   freezeHead(); freezeLast(READ(last))
28 }
29 @tailrec def freezeHead() {
30   val p = READ(head)
31   if (p >= 0) if (!CAS(head, p, -p - 1))
32     freezeHead()
33 }
34 @tailrec def freezeLast(p: Int) =
35   if (p >= 0 && p < array.length)
36     if (!CAS(array(p), EMPTY, FROZEN))
37       freezeLast(findLast(p))
```

Figure 5. Single-Shot Queue Freeze Operation

negative, the queue is frozen (explained in Section 2.2). If `p` is within bounds, `deq` reads the element at `p` to a local variable `x` in line 18. If `x` is either `EMPTY` or `FROZEN`, then the end of the queue was reached, and `deq` returns `NONE`. Otherwise, `deq` atomically increments `head` in line 20, or retries the operation when CAS fails. If the CAS in line 20 succeeds, the thread that executed it must eventually remove the element from `array` in line 21, to avoid memory leaks.

Lemma 2. *If `deq` returns NONE, the queue is either empty or frozen. If `deq` returns an element, the CAS operation in `deq` previously incremented `head` by one.*

2.2 Freeze Operation

Single-shot queue also exposes the `freeze` operation, which prevents subsequent updates from succeeding. The method `freeze` first calls `freezeHead`, which atomically replaces the `head` value with a corresponding negative value. This prevents subsequent `deq` operations. The `freeze` method then calls `freezeLast`, which enqueues a `FROZEN` element. This prevents subsequent `enq` operations.

After the `freeze` method completes, the `Segment` object becomes immutable – no operation will subsequently change its state. A stale write may change the `last` field, but the logical state is only defined by the `array` and `head`.

Lemma 3. *After `freeze` returns, subsequent operations can only modify the `last` field and the `array` entries preceding the position `-head - 1`.*

```
class SnapQueue[T] {
  abstract class Node
  class Frozen(val f: Trans, val root: Node)
    extends Node
  abstract class NonFrozen extends Node
  class Root(
    @volatile var left: Side,
    @volatile var right: Side)
    extends NonFrozen
  class Side(
    val isFrozen: Boolean,
    val segment: Segment[T],
    val support: Support[T])
  type Trans = NonFrozen => NonFrozen
  @volatile var root: Node = _
}
```

Figure 6. SnapQueue Data Types

Lemma 4. *When `freeze` returns, `array` either contains no `EMPTY` entries, or contains a single `FROZEN` entry. In both cases, `head` is negative.*

Note that, due to the `findLast` call, complexity of the `freeze` operation is $O(L)$, where L is the length of the array.

3. SnapQueue

Single-shot queue, or segment, is simple, and consequently efficient. However, boundedness and the L-operations limit make it uninteresting for most practical purposes. Our goal is an unbounded queue with arbitrary atomic operations, which we name SnapQueue. In this section, we show its data types.

When SnapQueue size is less than L, the elements are stored as a segment. SnapQueue overcomes segment's L-operations limit by reallocating the segment when it becomes full. To overcome boundedness, SnapQueue uses a secondary representation: a *segment-support pair*. When the number of elements exceeds L, the segment is replaced by two segments `left` and `right`. Subsequent `enqueue` operations use the `right` segment, and `dequeue` operations use the `left` segment.

If there are more than $2L$ elements, additional segments are stored in *support* data structures, hence the name segment-support pair. The support data structure is persistent and kept in the `support` field. Support data structure must be such that pushing and removing segments retains the order in which the segments were pushed, i.e. a sequence.

The `SnapQueue[T]` class is shown in Figure 6. It has a volatile field `root` of the `Node` type. `Node` is a supertype of `Segment` from Section 2, and of the `Root` and `Frozen` types. `Segment` and `Root` have another common supertype called `NonFrozen`. The `Root` comprises two volatile fields `left` and `right`, pointing to immutable segment-support pairs of type `Side`. Their `isFrozen` field denotes whether `Side` is frozen (explained in Section 3.1). The `support` field stores a persistent data structure of the `Support[T]` type, which stores intermediate segments. SnapQueue correctness

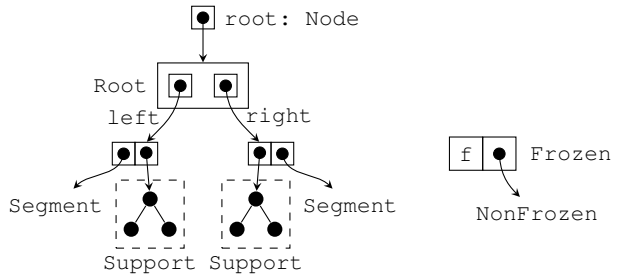

Figure 7. SnapQueue Illustration

does not depend on the choice of this data structure, so we defer discussion on `Support[T]` until Section 4. A particular SnapQueue instance containing a `Root` is shown in Figure 7.

The `Frozen` data type contains a reference to either a `Root` or a `Segment`, and denotes that the underlying SnapQueue is currently in the process of being frozen, or is already frozen. It also contains a transition function of type `NonFrozen => NonFrozen`.

As hinted earlier, SnapQueue uses an atomic transition operation, which freezes the SnapQueue before replacing its contents. In the next section, we study how freezing works.

3.1 Freeze Operation

Instead of starting with the basic queue operations as in Section 2, we first describe the `freeze` operation, which is a prerequisite for the `transition` operation, shown later.

As with the single-shot queue, invoking `freeze` turns a concurrent data structure into an immutable data structure. The goal of `freeze` is to invalidate subsequent writes on volatile fields in the SnapQueue. To do this, `freeze` first freezes the root, then freezes the two `Side` references, and at the end freezes the underlying segments.

The `SnapQueue#freeze` method in Figure 8 takes the reference `r` to the previous SnapQueue root. It also takes the transition function, which is explained in Section 3.2 – for now we treat it as extra payload in the `Frozen` object.

The `freeze` operation first attempts to replace the expected root `r` with a fresh `Frozen` object. If some other thread already changed `root` to a value different than `r`, `freeze` signals failure by returning `null`. Otherwise, if the `CAS` changes `root` to the `Frozen` object, `completeFreeze` is called to freeze the `left` and `right` side, and the segments.

Note that, unlike `freeze` on the single-shot queue from Section 2.2, `freeze` operation in Figure 8 may fail and return `null`. In this case, it is up to the caller to retry.

However, if the CAS in line 42 succeeds, the SnapQueue is eventually frozen, since other operations will never modify frozen `left` and `right` fields, or frozen segments.

Lemma 5. *After `freeze` returns a non-`null` value, subsequent operations may only modify the `last` fields and the `array` entries preceding `-head - 1` in the two segments.*

```
38 @tailrec
39 def freeze(r: NonFrozen, f: Trans) = {
40   val fr = new Frozen(f, r)
41   if (READ(root) != r) null
42   else if (CAS(root, r, fr)) {
43     completeFreeze(fr.root)
44     fr
45   } else freeze(r, f)
46 }
47 def completeFreeze(r: NonFrozen) =
48   r match {
49     case s: Segment => s.freeze()
50     case r: Root =>
51       freezeLeft(r)
52       freezeRight(r)
53       READ(r.left).segment.freeze()
54       READ(r.right).segment.freeze()
55   }
56 @tailrec def freezeLeft(r: Root) {
57   val l = READ(r.left)
58   if (l.isFrozen) return
59   val nl = new Side(
60     true, l.segment, l.support)
61   if (!CAS(r.left, l, nl)) freezeLeft(r)
62 }
```

Figure 8. SnapQueue Freeze Operation

Furthermore, if a segment is frozen, then so are the `left` and `right` fields.

Proof. This follows from Lemma 3, and the fact that basic SnapQueue operations fail if `Side` is frozen. □

Lemma 6. *If `freeze` returns a non-`null` value `fr`, then the value of `root` atomically changed from `r` to `fr`, where `r` is the specified root reference.*

The complexity of the `freeze` operation in this Section is again $O(L)$, where L is the length of the array in `Segment` objects. Importantly, complexity does not depend on the size of the persistent `Support` data structure.

3.2 Transition Operation

The `transition` operation is the most important SnapQueue operation. It atomically exchanges the contents and the structure of the SnapQueue. It is analogous to the CAS operation, with the difference that `transition` operates on the entire data structure, rather than a single memory location.

The `transition` operation first freezes the SnapQueue. If `freeze` fails, then so does `transition`. However, if freezing succeeds, `transition` uses a *transition function* that maps a frozen SnapQueue into a new SnapQueue.

The `SnapQueue#transition` method in Figure 9 takes the expected root value `r`, and a transition function `f`. After freezing, it invokes `completeTransition`, which computes the new root with `f`, and atomically replaces the old root with the output of `f` in line 79. The `helpTransition` method is

```
63 def transition(r: Node, f: Trans):
64   NonFrozen = {
65   if (r.isFrozen) {
66     completeTransition(r)
67     null
68   } else {
69     val fr = freeze(r, f)
70     if (fr == null) null
71     else {
72       completeTransition(fr)
73       fr.root
74     }
75   }
76 }
77 def completeTransition(fr: Frozen) {
78   val nr = fr.f(fr.root)
79   while (READ(root) == fr) CAS(root, fr, nr)
80 }
81 def helpTransition() {
82   READ(root) match {
83     case fr: Frozen =>
84       completeFreeze(fr.root)
85       completeTransition(fr)
86     case _ => // not frozen -- do nothing
87   }
88 }
```

Figure 9. SnapQueue Transition Operations

similar, but gets called by threads executing their separate operations, to help complete the transition in a lock-free way.

Lemma 7. *If `transition` returns a non-`null` value, then the value of `root` atomically changed from `Node` reference `r` to `Frozen(r, f)`, and then atomically to `f(r)`, where `r` is the specified root and `f` is the transition function.*

Proof. This is a consequence of Lemmas 5 and 6. □

Lemma 8. *The `transition` running time is $O(L+f(n,L))$, where $O(f(n,L))$ is the transition function running time with respect to the support data structure size n, and segment length L.*

Proof. This is a consequence of the $O(L)$ complexity of the `freeze` operation, and the fact that the transition function is invoked at least once. □

3.3 Transition Functions

The transition operation takes a referentially transparent transition function. The transition function is invoked by the time the `Node` is already frozen, and effectively a persistent data structure, Therefore, the transition can be reasoned about functionally, separate from concurrency concerns.

We consider some concrete transition functions in Figure 10. Assume that the SnapQueue represented with a segment of length L needs to be replaced with a `Root`, as outlined at the beginning of Section 3. If the SnapQueue contains

```
89 def expand(r: NonFrozen) = r match {
90   case s: Segment =>
91     val head = s.locateHead
92     val last = s.locateLast
93     if (last - head < s.array.length / 2) {
94       copy(s)
95     } else new Root(
96       new Side(false, unfreeze(s), create()),
97       new Side(false, new Segment, create()))
98 }
99 def transfer(r: NonFrozen) = r match {
100   case r: Root =>
101     if (r.right.support.nonEmpty) new Root(
102       new Side(false,
103         copy(READ(r.left).segment),
104         READ(r.right).support),
105       new Side(false,
106         copy(READ(r.right).segment),
107         create()))
108     else copy(READ(r.right).segment)
109 }
```

Figure 10. SnapQueue Transition Functions

less than $L/2$ elements, the `expand` transition function creates another segment – `copy` in line 94 copies a frozen segment into a freshly allocated one. If the size is above $L/2$, `expand` creates a `Root` object – here, `create` allocates an empty support data structure, and `unfreeze` reallocates the segment. The $L/2$ approach prevents eagerly alternating between `Root` and `Segment` representations.

The `transfer` transition function is used when a `Root` queue runs out of elements in its `left` segment and support structure. It transfers the support structure from the `right` to the `left` side of the SnapQueue. In Section 3.4, we will see usages of both `transfer` and `expand`.

Lemma 9. *The running time of both `expand` and `transfer` is $O(L)$, where L is the segment size.*

Proof. This is the consequence of calling `unfreeze` and `copy`, which take $O(L)$ time for an L-element segment. □

3.4 Basic Operations

In this section, we closely examine enqueue and dequeue. For efficiency, SnapQueue basic operations in most cases only modify a segment. Occasionally, a segment is used up and replaced with a new segment from the support structure.

We define the top-level `SnapQueue#enqueue` method, and three internal, dynamically dispatched `enqueue` methods on the `Root`, `Frozen` and `Segment` data types. After the top-level `enqueue` in Figure 11 reads the `root`, it calls an internal `enqueue`. Since SnapQueue is an unbounded data structure, it must always be possible to enqueue a new element. If an internal `enqueue` returns `false` to indicate that the element was not added, the operation is retried.

The `enqueue` on the `Root` type reads the `right` segment-support pair, and the segment's `last` field. It then calls `enq`

```
110 // SnapQueue
111 @tailrec def enqueue(x: T): Unit =
112   if (!READ(root).enqueue(x)) enqueue(x)
113 // Root
114 @tailrec def enqueue(x: T): Boolean = {
115   val r = READ(right)
116   val p = READ(r.segment.last)
117   if (r.segment.enq(p, x)) true
118   else { // full or frozen
119     if (r.frozen) false
120     else { // full
121       val seg = new Segment
122       val sup = pushr(r.segment, r.support)
123       val nr = new Side(false, seg, sup)
124       CAS(right, r, nr)
125       enqueue(x)
126     }
127   }
128 }
129 // Frozen
130 def enqueue(x: T): Boolean = {
131   helpTransition()
132   false
133 }
134 // Segment
135 def enqueue(x: T): Boolean = {
136   val p = READ(last)
137   if (enq(p, x)) true
138   else {
139     if (READ(head) < 0) false // frozen
140     else { // full
141       transition(this, expand)
142       false
143     }
144   }
145 }
```

Figure 11. SnapQueue Enqueue Operation

```
146 // SnapQueue
147 @tailrec def dequeue(): T = {
148   val r = READ(root)
149   val x = r.dequeue()
150   if (x != REPEAT) x else dequeue()
151 }
152 // Root
153 def dequeue(): T = {
154   val l = READ(left)
155   val x = l.segment.deq()
156   if (x != NONE) x
157   else { // empty or frozen
158     if (l.frozen) REPEAT
159     else { // empty
160       if (l.support.nonEmpty) {
161         val (seg, sup) = copy(popl(l.support))
162         val nl = new Side(false, seg, sup)
163         CAS(left, l, nl)
164         dequeue()
165       } else { // empty side
166         transition(this, transfer)
167         REPEAT
168       }
169     }
170   }
171 }
172 // Frozen
173 def dequeue(): T = {
174   helpTransition()
175   REPEAT
176 }
177 // Segment
178 def dequeue(): T = {
179   val x = deq()
180   if (x != NONE) x
181   else if (READ(head) < 0) REPEAT // frozen
182   else NONE // empty
183 }
```

Figure 12. SnapQueue Dequeue Operation

in line 117. If `enq` returns `false`, the segment is either full or frozen, by Lemma 1. In line 119, we rely on Lemmas 4 and 5 to check if the segment is frozen. By Lemma 5, if the segment is frozen, the `Root` is also frozen, so we return to the top-level `enqueue` in line 119, and then help complete the transition. Otherwise, if the segment is full, we push its array into the support structure with a persistent `pushr` operation, and replace the `right` side with the new `Side` object.

The `enqueue` on the `Frozen` type helps the current transition operation to complete, and then retries the `enqueue`. The `enqueue` on the `Segment` type calls `enq` – if the segment is full, it attempts to replace the segment by calling the `transition` operation from Section 3.2 with the `expand` function from Section 3.3 before retrying.

The `dequeue` operation in Figure 12 is similarly separated between the top-level `SnapQueue#dequeue`, and internal `dequeue` methods on `Root`, `Segment` and `Frozen` types. `Root#dequeue` starts by calling `deq` on the `left` segment. If `deq` fails, the segment is either empty or frozen, by Lemma 2. If the segment is frozen, there is possibly an ongoing transition, so control returns to the top-level `dequeue`. Otherwise, if the support data structure is non-empty, `dequeue` invokes `popl` to extract an array for a new segment, refreshes the `left` side, and retries. If the support is empty, segments must be borrowed from the `right` side, so `transition` with the `transfer` function from Section 3.3 is called.

We can now state the following two theorems. The first theorem establishes the operation running time.

Theorem 1 (SnapQueue Running Time)**.** *Let L be the length of the segment `array`. The amortized running time of the enqueue operation is $O(L + \frac{f(n)}{L})$, where $O(f(n))$ is the running time of the `pushr` operation of the support structure containing n elements. The amortized running time of the dequeue operation is $O(L + \frac{g(n)}{L})$, where $O(g(n))$ is the running time of the `popl` operation of the support structure containing n elements.*

The second theorem establishes the contention rate between producers and consumers as the number of operations

```
184 def id(r: NonFrozen) = r match {
185   case s: Segment => copy(s)
186   case r: Root => new Root(
187     new Side(false, unfreeze(r.left.segment),
188       r.left.support),
189     new Side(false, copy(r.right.segment),
190       r.right.support))
191 }
192 @tailrec def snapshot() = {
193   val r = READ(root)
194   val nr = transition(r, id)
195   if (nr == null) {
196     helpTransition()
197     snapshot()
198   } else id(nr)
199 }
```

Figure 13. SnapQueue Snapshot

```
200 @tailrec def concat(that: SnapQueue[T]) = {
201   if (stamp(this) < stamp(that)) {
202     val p = new Promise[NonFrozen]
203     val r = READ(this.root)
204     val nr = this.transition(r, rthis => {
205       if (!p.isCompleted) {
206         val rthat = that.snapshot()
207         val res = concatenate(rthis, rthat)
208         p.trySuccess(res)
209       }
210       id(rthis)
211     })
212     if (nr == null) this.concat(that)
213     else new SnapQueue(p.getValue)
214   } else { /* analogous */ }
215 }
```

Figure 14. SnapQueue Concatenation

between writes to the `root` field – the higher this value, the lower the contention. When the consumer periodically exhausts the support data structure in the left `Side` object, the `root` is frozen to call `transfer`. When this happens, the producers and consumers contend. We care about this, since typical actor systems have a single consumer and many producers – for example, multiple producers should not be able to flood the single consumer as a consequence of contention.

Theorem 2 (SnapQueue Contention)**.** *Let L be the length of the segment `array`, $O(g(n))$ the running time of the `popl` operation, and n the total number of elements in the SnapQueue. In a sequence of basic operations, there are on average $O(L + \frac{n \cdot g(n)}{L})$ dequeue operations between two `root` field writes (that is, between two `freeze` operations).*

Proof. Both theorems are a consequence of Lemmas 8 and 9, and implementations in Figures 11 and 12. □

3.5 Snapshot and Concatenation

The `transition` operation from Section 3.2 is very expressive, as it atomically changes the state of the SnapQueue given an arbitrary transformation. In this section, we use it to implement atomic snapshots and atomic concatenation.

The `snapshot` method in Figure 13 uses the *identity* transition function `id`. This method repetitively invokes the `transition` operation until it becomes successful. It is easy to see that the `snapshot` running time is $O(L)$.

Next, we note that the Lemma 7 implies the following:

Lemma 10. *If the value returned by the transition function `f` gets written to the `root` field by `transition`, then during the corresponding invocation of `f`, SnapQueue was frozen.*

We rely on Lemma 10 to implement atomic concatenation. To achieve atomicity, this operation must simultaneously freeze two SnapQueues. Note that two concatenation operations on the same pair of SnapQueues could potentially deadlock if they freeze the SnapQueues in the opposite orders. To prevent this, we need to establish an ordering between SnapQueue instances – we assume that a `stamp` method associates unique integers to each queue.

The `SnapQueue#concat` operation in Figure 14 first calls `stamp` to establish the order. We assume `this` comes before the argument `that` – the reverse case is analogous. The `concat` method creates a new `Promise` object [9], which serves as a placeholder for the resulting SnapQueue. The `Promise` object is a *single-assignment variable* – it can be assigned a value at most once using the `trySuccess` method. After `concat` starts a `transition` on `this` SnapQueue, it creates a snapshot of `that` SnapQueue. At this point, we have two frozen data structures and can concatenate them with a persistent `concatenate` operation in line 207. The result is stored into the `Promise` in line 208.

Assume that the `transition` returns a non-`null` value. This can only happen if some thread assigned the result into the `Promise` before `transition` returned. That thread called `snapshot` on `that` before writing to the `Promise`. By Lemma 10, `this` SnapQueue was frozen at the time, implying that the concatenation operation is atomic with respect to both SnapQueues.

4. Support Data Structures

In Section 3, we introduced SnapQueue, which relies on a persistent sequence data structure, called *support*, to store arrays of size L. Since the running time of SnapQueue operations depends on the running time of the support data structure, SnapQueue constitutes a framework for assessing different persistent data structure implementations. In this section, we consider support structures that provide the following operations: `create`, which creates an empty support structure, `nonEmpty`, which checks if the support structure is empty, `pushr`, which appends the element on the right side, `popl`, which removes the leftmost element, and `concatenate`, which concatenates two support structures.

```
trait Conc[+T] {
  def level: Int
  def size: Int
  def normalized = this
}

case object Empty extends Conc[Nothing] {
  def level = 0
  def size = 0
}

case class <>[T](left: Conc[T], right: Conc[T])
extends Conc[T] {
  val level = 1 + max(left.level, right.level)
  val size = left.size + right.size
}

case class Single[T](x: T) extends Conc[T] {
  def level = 0
  def size = 1
}
```

Figure 15. Conc-Tree Data Types

When choosing a data structure, we usually consider the following properties. First, *asymptotic running time* with respect to the data structure size should be as low as possible, in the ideal case $O(1)$. Second, *constant factors* must be low to ensure good absolute running time. With high constant factors, even an $O(1)$ operation can be prohibitively slow for typical data structure sizes. Finally, the data structure should be *simple* – it should be easy to comprehend and implement.

Although persistent data structures with worst-case $O(1)$ implementations of the desired operations exist [12], their implementations are not simple, and have high constant factors. We know from Theorem 1 that the SnapQueue operations run in $O(L + \frac{f(n)}{L})$ time. To optimize SnapQueues, we must optimize L against the support structure whose `pushr` and `popl` operations are $O(1)$ with low constant factors.

In the rest of this section, we study Conc-Trees [17] as the support data structure. Conc-Tree is a particular implementation of the *conc-list* abstraction from Fortress [5] [25], originally intended for functional task-parallel and data-parallel programming. Conc-Tree concatenation is $O(\log n)$, but its $O(1)$ `popl` and `pushr` are simple and efficient.

The `Conc` abstract data type is shown in Figure 15. This data type specifies the `size` of the tree, i.e. the number of elements, and the `level`, i.e. the tree height. A `Conc` is either an `Empty` tree, a tree with a `Single` element, or an inner node `<>` (pronounced *conc*) with two subtrees. `Conc` defines a method `normalized`, which returns the tree composed only of these three data types.

Conc-Trees maintain the following invariants. First, the `Empty` tree is never a child of other nodes. Second, the absolute `level` difference of the children in `<>` nodes is smaller than or equal to 1. This ensures balanced Conc-Trees – longest and shortest paths differ by at most $2\times$.

```
implicit class ConcOps[T](xs: Conc[T]) {
  def <>(ys: Conc[T]) = {
    if (xs == Empty) ys
    else if (ys == Empty) xs
    else conc(xs.normalized, ys.normalized)
  }
}
def conc[T](xs: Conc[T], ys: Conc[T]) = {
  val diff = ys.level - xs.level
  if (abs(diff) <= 1) new <>(xs, ys)
  else if (diff < -1) {
    if (xs.left.level >= xs.right.level) {
      val nr = conc(xs.right, ys)
      new <>(xs.left, nr)
    } else {
      val nrr = conc(xs.right.right, ys)
      if (nrr.level == xs.level - 3) {
        val nr = new <>(xs.right.left, nrr)
        new <>(xs.left, nr)
      } else {
        val nl = new <>(xs.left, xs.right.left)
        new <>(nl, nrr)
      }
    }
  } else { /* analogous */ }
}
```

Figure 16. Conc-Tree Concatenation

In Figure 16, we implement concatenation for Conc-Trees. Similar to how the symbol of the Scala `::` data type (pronounced *cons*) is used to prepend to a `List`, we borrow the `<>` data type symbol for `Conc` concatenation. Thus, the expression `new <>(xs, ys)` links the two trees `xs` and `ys` by creating a new `<>` object, whereas `xs <> ys` creates a balanced Conc-Tree that is a concatenation of `xs` and `ys`.

The public `<>` method eliminates `Empty` trees before calling the recursive `conc` method. The `conc` method links trees if the invariants allow it. If they do not, `conc` concatenates the smaller Conc-Tree with a subtree in the bigger Conc-Tree, before re-linking the result. The `conc` running time is $O(|h_{xs} - h_{ys}|)$, where h_{xs} and h_{ys} are the tree heights [17].

The `<>` method is sufficient for `pushr`:

```
def pushr(xs: Conc[T], x: T) = xs <> Single(x)
```

Unfortunately, this implementation is $O(\log n)$, and our goal is to achieve constant time. We will next extend the basic Conc-Tree data structure to achieve this goal.

In Figure 17, we introduce a new type `Ap`, which is isomorphic to the `<>` data type. The distinction with the `Ap` type is that its subtrees do not need to differ in `level` by at most 1. Instead, `Ap` introduces two new invariants. First, an `Ap` node can only be the left subtree of another `Ap` node. Otherwise, an `Ap` must be the root of the tree. Second, if an `Ap` node `n` has a `Ap` node `m` in its left subtree, then `n.right.level` must be strictly smaller than `m.right.level`.

As a consequence of the two `Ap` invariants, Conc-Trees that contain `Ap` nodes correspond to to numbers in the binary

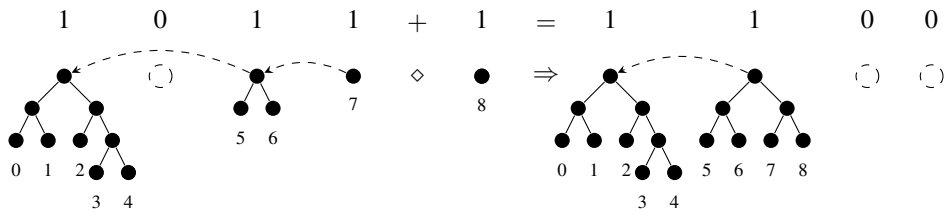

Figure 18. Correspondence Between the Binary Number System and Conc-Trees

```
case class Ap[T](left: Conc[T], right: Conc[T])
extends Conc[T] {
  val level = 1 + left.level.max(right.level)
  val size = left.size + right.size
  override def normalized = wrap(left, right)
}
def wrap[T](xs: Conc[T], ys: Conc[T]) =
  xs match {
    case Ap(ws, zs) => wrap(ws, zs <> ys)
    case xs => xs <> ys
  }
```

Figure 17. Conc-Tree Append Data Type

```
def append[T](xs: Ap[T], ys: Conc[T]) =
  if (xs.right.level > ys.level) new Ap(xs, ys)
  else {
    val zs = new <>(xs.right, ys)
    xs.left match {
      case ws @ Ap(_, _) => append(ws, zs)
      case ws =>
        if (ws.level <= xs.level) ws <> zs
        else new Ap(ws, zs)
    }
  }
```

Figure 19. Conc-Tree Append Operation

system, as shown in Figure 18. Each `Ap` node `n` corresponds to a digit whose position is equal to the height `n.right`. Absence of an `Ap` node with a specific height indicates the absence of the corresponding digit.

This correspondence has two consequences. First, since we know how to increment a binary number by adding a single digit on the right, we can equivalently append `Single` trees by linking trees of the same height together [16]. Second, since incrementing a number in the binary number system on average takes $O(1)$ computational steps, appending a `Single` tree also takes $O(1)$ steps on average.

This is illustrated in Figure 18, where a `Single` tree is appended to a Conc-Tree corresponding to the binary number 1011. Appending triggers a chain of carries, which stops upon reaching 0-digit. The `append` method that implements this is shown in Figure 19, and it is used to directly implement the `pushr` operation:

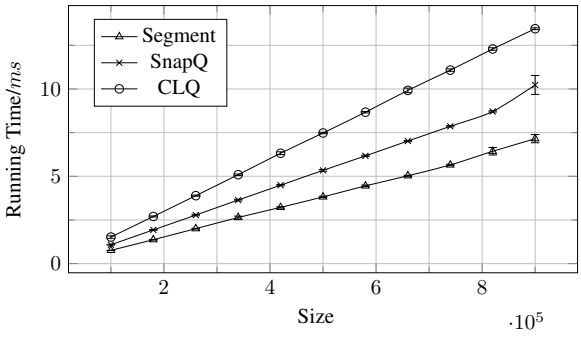

Figure 20. 1-Thread Enqueue, L = 128

```
def pushr(xs: Ap[T], x: T): Ap[T] =
  append(xs, Single(x))
```

The `append` implementation is similar to functional *random access lists* [16]. Both achieve amortized $O(1)$ appends using a number system representation. However, Conc-Trees with `Ap` nodes can additionally be concatenated in $O(\log n)$ time. For this, an `Ap` tree must first be `normalized` – transformed into a tree composed only of `Single` and `<>` nodes. This is the task of the `wrap` method in Figure 17, which concatenates all the `right` subtrees in the append list. Since concatenation takes $O(|h_{xs} - h_{ys}|)$ time, and the sum of consecutive height differences between the `right` subtrees is $O(\log n)$, the `wrap` method runs in $O(\log n)$ time [17].

To conclude, we defined a persistent data structure with amortized $O(1)$ `pushr`, and $O(\log n)$ concatenation. Although we did not show it in this section, we can similarly define a `Prep` node to obtain a $O(1)$ `popl`.

5. Evaluation

In this section, we experimentally evaluate optimal segment lengths, and compare the SnapQueue against the JDK `ConcurrentLinkedQueue` (CLQ) implementation based on Michael and Scott's concurrent queue [14]. The benchmarks are performed on an Intel 2.8Ghz i7-4900MQ quad-core processor with hyperthreading. We use the ScalaMeter tool to execute our benchmarks [18], which relies on standard JVM benchmarking methodologies [8].

The workload in the following benchmarks comprises of either enqueuing or dequeuing $size$ elements, spread across some number of threads. When we study single-thread performance, we vary the $size$ on the x-axis. Otherwise, we

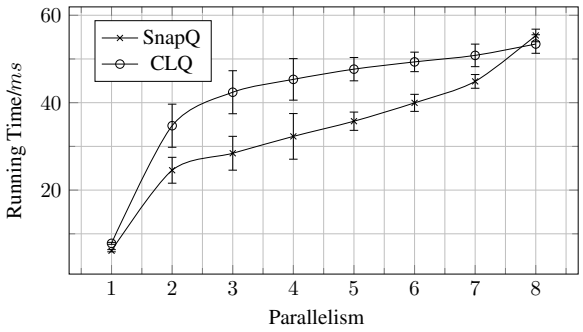

Figure 21. N-Thread Enqueue, size = $500k$, L = 128

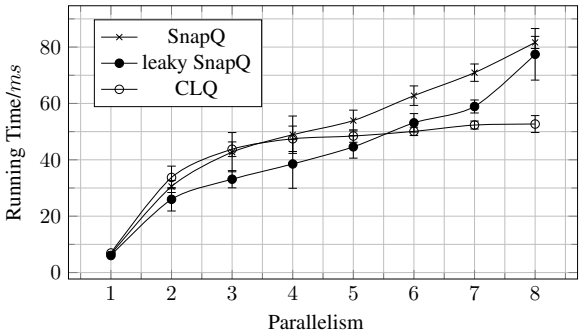

Figure 23. N-Thread Dequeue, size = $500k$, L = 128

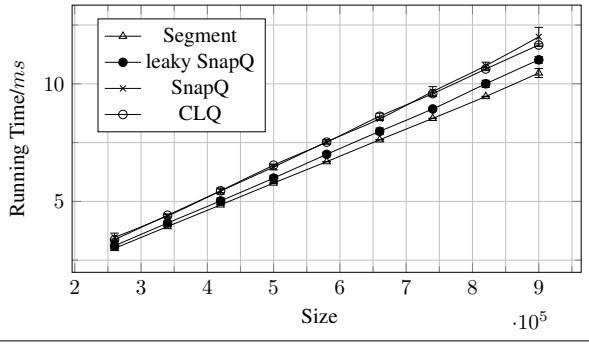

Figure 22. 1-Thread Dequeue, L = 128

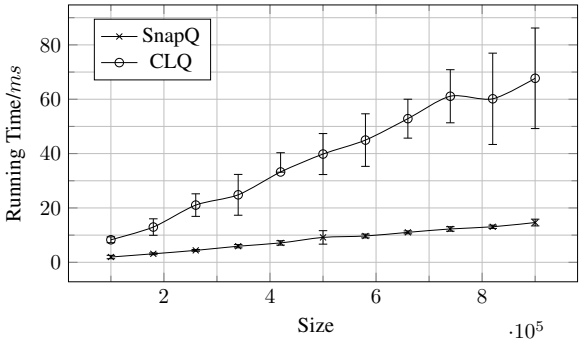

Figure 24. 1 Producer, 1 Consumer

vary the number of threads and keep the total number of elements $size$ fixed. The y-axis shows the running time. The benchmarks test the two data structures in extreme conditions. In practical applications, producers and consumers perform some work between `enqueue` and `dequeue` calls, but here they exclusively call queue operations.

In Figure 20, we compare the single-thread running time of SnapQueue `enqueue` method against the running time of the `add` method of CLQ. The `enq` method of the `Segment` class is almost $2\times$ smaller than that of CLQ. This speedup is due to the extra cost of allocating a `Node` object that CLQ pays when enqueuing an element. With $L = 128$, SnapQueue `enqueue` performance is around 25% better than CLQ – the performance loss with respect to `Segment` is due to additional atomic reads of `root` and `right` fields, intermediate method calls, and Conc-Tree manipulations.

In Figure 21, we show the same workload distributed across P threads, where P varies from 1 to 8. This benchmark shows that CLQ contention is larger than SnapQueue contention for fewer threads, but the two data structures have similar contention at higher parallelism levels.

Next, we show single-thread `dequeue` performance in Figure 22. Here, SnapQueue and CLQ have almost identical running times – CLQ's `poll` method is faster than the `add` method from Figure 20, since `poll` does not have to allocate any objects. Calling `deq` on `Segment` is about 15% faster.

The *leaky SnapQ* is a version of SnapQueue with a slightly faster `dequeue` operation. In line 21 of the listing in Figure 4, we used an atomic WRITE operation to replace the logically removed element with a special `REMOVED` value. This ensures that the SnapQueue does not keep references to dequeued elements in the leftmost segment, and avoids a potential memory leak. The WRITE in the `dequeue` operation mutates arrays that come from the support data structure. Since arrays in the support data structure, due to operations such as `snapshot`, can be shared between multiple copies of the SnapQueue, `dequeue` must copy the array returned by `popl` in line 161 of the listing in Figure 12.

Some applications are unaffected by memory leaks, either because the objects are small, or because the `dequeue` operation is called often. In such applications, the WRITE in line 21 is unnecessary. Consequently, the `copy` call in line 161 can be removed. This improves performance of the `dequeue` operation by about 10%, as shown in Figures 22 and 23.

In Figure 23, we show the same workload spread across multiple threads. The contention effects are similar between SnapQueue and CLQ when $P \leq 4$. For $P > 4$, SnapQueue contention-related slowdowns are larger than those of CLQ. We believe that this is due to increased amount of allocations that occur when multiple threads simultaneously attempt to replace the leftmost segment after the `popl` in line 161.

Next, we test for contention between a single producer and a single consumer in Figure 24. Here, the producer starts by enqueuing $size$ elements, and the consumer simultaneously dequeues them. While in this benchmark SnapQueue shows almost the same performance as in Figure 22, CLQ

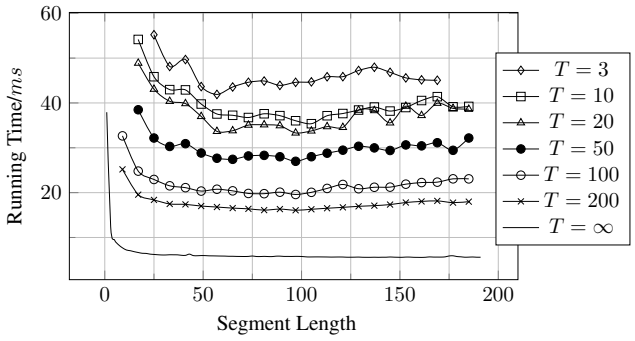

Figure 25. Running Time vs Segment Length

suffers from contention. We postulate that this is because the CLQ `poll` method is faster than the `add` method, so the consumer repeatedly catches up with the producer, causing contention. This effect is important only for those applications in which producers and consumers spend most of the time invoking queue operations.

For the purposes of the last benchmark, we recall the Theorem 1, where we established that the running time of queue operations is bound by $O(L + \frac{f(n)}{L})$, where L is the segment length, and $O(f(n))$ is the complexity the support data structure operations. Every queue operation can be interrupted with a snapshot, in which case two segments of length L must be copied, hence the first term. Then, after every L operations, a segment must be pushed or popped from the support structure, hence the second term $\frac{f(n)}{L}$. Assuming that snapshots occur periodically at some rate T, the running time becomes $O(\frac{L}{T} + \frac{f(n)}{L})$. Optimizing the SnapQueue amounts to optimizing this expression for some period T.

Figure 25 shows the dependency between the running time of the `enqueue` method and the SnapQueue segment length for different choices of the period T. For $T = \infty$, that is, when snapshots do not occur, the first term becomes 0 and the corresponding curve converges around $L = 50$, but has no optimum. For other values of T, the optimum value appears somewhere between $L = 64$ and $L = 128$. For $T = 3$, that is, when snapshots occur very frequently, the first term becomes dominant and the optimum shifts to around $L = 64$. Based on these results, the recommended SnapQueue `Segment` length is 64 for most applications.

6. Related Work

Lock-free concurrent data structures are an active area of research, and there exist extensive concurrent data structure overviews [10] [15]. Here we focus on the related concurrent queues and data structures with snapshots.

Original lock-free concurrent queues allocate a separate node per each enqueued object [14]. SnapQueue avoids this by allocating a chunk of memory (segment), and stores the objects within. As a result, SnapQueue has decreased memory footprint compared to traditional lock-free queues.

The single-shot lock-free queue is similar to the FlowPool data structure [20], which also allocates memory chunks instead of single nodes per element. FlowPool basic operations use two CAS operations instead of one, and are more complex. FlowPools can also reduce contention at the cost of weakened ordering guarantees [23], but do not have atomic snapshots. Disruptor [27] is bounded concurrent ring-buffer data structure designed to serve as a high-performance buffer. Its memory is also allocated in chunks for better cache locality, and decreased memory footprint.

Some concurrent data structures have in the past provided efficient snapshots. SnapTree [6] is a concurrent AVL tree that provides a fast clone operation and consistent iteration. Ctrie [19] [20] is a concurrent hash trie implementation with constant time atomic lazy snapshots, atomic clear operation and consistent iteration. Similar to how SnapQueue uses a persistent data structure to efficiently switch between global states, Ctrie relies on a persistent data structure to implement its lazy snapshot operation. A different approach in the past was to provide a general framework for snapshots [4] – although this is a general lock-free snapshot technique, these snapshots are generally $O(n)$ in the size of the data structure.

An extensive overview of traditional persistent data structure design techniques is given by Okasaki [16]. Binary number representation used by Conc-Trees is inspired by *random access lists*, which represent data as a list of complete binary trees. In this regard, Conc-Trees are unlike most traditional immutable sequence data structures, where every binary digit of weight W corresponds to a *complete tree* with W elements, but instead rely on relaxed balancing typically used in AVL trees [3]. This combination of features allow Conc-Trees to retain logarithmic concatentation along with amortized constant time prepend and append operations.

Conc-list abstraction appeared in Fortress [5], where it was used for task-parallel programs [25]. More advanced Conc-Tree variants achieve worst-case $O(1)$ prepend and append [17], but are more complicated. Some persistent trees achieve amortized $O(1)$ [11] and worst-case [12] $O(1)$ concatenation, at the cost of high implementation complexity.

7. Conclusion

We presented the SnapQueue data structure – a lock-free, concurrent queue implementation with atomic global transition operation. This transition operation is the building block for enqueue and dequeue operations, as well as atomic snapshots and concatenation. We note that transition operation is not limited to these applications, but can be used for e.g. atomic clear, size retrieval or a reverse operation. SnapQueue stores its state in a persistent support data structure, and the transition operation leverages this to improve performance.

Although we did not explicitly show lock-freedom, we note that lock-freedom can be easily proved by showing that each failing CAS implies the success of another concurrent

operation, and that each state change is a finite number of instructions apart.

We analyzed the running time and contention. This analysis shed light on several important design ideas. First, the contention between the producers and the consumers is an inverse function of the number of elements in the queue. Second, the cost of periodic segment reallocation is amortized by the segment length L – optimizing the SnapQueue is a matter of finding the optimal value for L. Third, SnapQueue performance depends on the underlying persistent support structure. We described Conc-Trees as a concrete support data structure example with $O(1)$ prepend and append, and $O(\log n)$ concatenation.

Most importantly, we showed that SnapQueue does not incur any performance penalties by having atomic snapshot operations. SnapQueue has similar, and in some cases better, performance compared to other concurrent queues.

This paper revealed several benefits of using persistent data structures when implementing global-state operations. We saw that, in conjunction with an atomic snapshot, it is sufficient to piggy-back the support structure to achieve the desired running time. This indicates that persistent data structures are not relevant only for functional programming, but also crucial for concurrent data structure design.

Acknowledgments

We would like to thank prof. Doug Lea from SUNY Oswego for his feedback and useful suggestions.

References

[1] Akka documentation, 2015. http://akka.io/docs/.

[2] SnapQueue Implementation, 2015. https://github.com/storm-enroute/reactive-collections/tree/master/reactive-collections-core.

[3] G. M. Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.

[4] Y. Afek, N. Shavit, and M. Tzafrir. Interrupting snapshots and the JavaTM size method. *J. Parallel Distrib. Comput.*, 72(7):880–888, July 2012.

[5] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[6] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45(5):257–268, Jan. 2010.

[7] C. Click. Towards a scalable non-blocking coding style, 2007.

[8] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, Oct. 2007.

[9] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic. Scala Improvement Proposal: Futures and Promises (SIP-14). 2012.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[11] R. Hinze and R. Paterson. Finger trees: A simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, Mar. 2006.

[12] H. Kaplan and R. E. Tarjan. Purely functional representations of catenable sorted lists. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 202–211, 1996.

[13] D. Lea. Doug Lea's workstation, 2014.

[14] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[15] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14 — 47–30, 2007. Chapman and Hall/CRC Press.

[16] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[17] A. Prokopec. *Data Structures and Algorithms for Data-Parallel Computing in a Managed Runtime*. PhD thesis, IC, EPFL, Lausanne, 2014.

[18] A. Prokopec. ScalaMeter Benchmarking Suite Website, 2014. http://scalameter.github.io.

[19] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. pages 151–160, 2012.

[20] A. Prokopec, H. Miller, T. Schlatter, P. Haller, and M. Odersky. FlowPools: A lock-free deterministic concurrent dataflow abstraction. In *LCPC*, pages 158–173, 2012.

[21] W. Pugh. Concurrent maintenance of skip lists. Technical report, College Park, MD, USA, 1990.

[22] W. N. Scherer, D. Lea, and M. L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, 2009.

[23] T. Schlatter, A. Prokopec, H. Miller, P. Haller, and M. Odersky. Multi-lane flowpools: A detailed look. Technical report, EPFL, Lausanne, September 2012.

[24] R. Soulé, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel. Dynamic expressivity with static optimization for streaming languages. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 159–170, New York, NY, USA, 2013. ACM.

[25] G. Steele. Organizing functional code for parallel execution; or, foldl and foldr considered slightly harmful. International Conference on Functional Programming (ICFP), 2009.

[26] S. Tasharofi. *Efficient testing of actor programs with non-deterministic behaviors*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.

[27] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. May 2011.