Faster Channels in Go (Work in Progress)

Scaling Blocking Channels with Techniques from Nonblocking Data-Structures.

Contents

Introduction

Channels in the Go language are a common way to structure concurrent code. The channel API in Go is intended to support programming in the manner described by CSP (see Hoare 1978, the original paper; also the preface of Donovan and Kernighan 2015 for CSP's relationship to Go). Channels in Go have a fixed buffer size b such that only b senders may return without having handed a value off to a corresponding receiver. Here is some basic pseudocode[1] for the send and receive operations[2], though it is worth referring to the spec ("The Go Programming Language Specification" 2009) as well.

```
send(c: chan T, item: T)
  atomically do:
    if the buffer is full
      block
    append an item to the buffer
    if there were any receivers blocked
      wake the first one up
```

```
receive(c: chan T) -> T
  atomically do:
      begin:
      if there are items in the buffer
        result = head of buffer
        advance the buffer head
        if there are any senders waiting
          wake the first sender up
        return result
      if the buffer is empty
        block
        goto begin
```

Go channels currently require goroutines[3] to acquire a single lock before performing additional operations[4]. This makes contention for this lock a scalability bottleneck; while acquiring a mutex can be very fast this means that only one thread can perform an operation on a queue at a time. This document describes the implementation of a novel channel algorithm that permits different sends and receives to complete in parallel.

We will start with a review of recent literature on non-blocking queues. Then we will move onto describing the implementation of a fast *unbounded* channel in Go; this algorithm may be of independent interest. Finally we will extend this design to provide the bounded semantics of Go channels. We will also report performance measurements for these algorithms.

Non-blocking Queues

The standard data-structure closest to the notion of unbounded channel is that of a FIFO queue. A queue supports enqueue and dequeue operations, where it is common for dequeue to be allowed to

[1] Psuedocode in this document will increasingly resemble real, working Go code. While we will try to explain core Go concepts as we go, a passing familiarity with Go syntax (or at least a willingness to squint and pretend one is reading C) will be helpful.

[2] Our focus is send and receive; we do not cover `select` or `close` here. `Close` would be fairly simple to add, `select` could be implement by using channels for the waiting mechanism used by receivers. While this would not be difficult, it would slow things down compared to the `WaitGroup` implementation.

[3] Go's standard unit of concurrency is called a goroutine. Goroutines take the place of threads in a language like C, but they are generally much cheaper to create and provide faster context switches. Many goroutines are independently scheduled on top of a smaller number of native operating system threads. This scheduling is not preemptive in the standard implementation, rather goroutines implicitly yield on function-call boundaries.

[4] See the Go channel source. In particular note calls to `lock` in `chansend` and `chanrecv`.

fail if there are no elements in the queue. There are myriad algorithms for concurrent queues which provide different guarantees in terms of progress and consistency (see M. Herlihy and Shavit 2008, chap. 10 for an overview), but we will focus here on *non-blocking* queues because of the approach in that literature to making scalable concurrent data-structures.

Informally, we say a data-structure is non-blocking if no thread can perform an operation that will require it to block any other threads for an unbounded amount of time. As a result, no queue that requires a thread to take a lock can be non-blocking: one thread can acquire a lock and then be de-scheduled for an arbitrary amount of time and thereby block all other threads contending for the lock. Non-blocking algorithms generally use atomic instructions like Compare-And-Swap (CAS) to avoid different threads stepping on one another's toes (see M. Herlihy and Shavit 2008, chap. 3 for a tutorial on atomic synchronization primitives). Non-blocking operations can exhibit a number of additional progress guarantees:

- **Obstruction Freedom** If there is only one thread executing an operation, that operation will complete in a finite number of steps.
- **Lock Freedom** Regardless of the number of threads executing an operation concurrently, at least one thread will complete the operation in a finite number of steps.
- **Wait Freedom** Any thread executing an operation is guaranteed to finish in a finite number of steps.

Non-blocking synchronization is not a panacea. The fact that there are hard upper bounds on how long it will take for a thread to complete an operation does not imply that the algorithm will perform better in practice. While wait-free data-structures are important for some embedded or real-time systems that need these strong guarantees, there are often blocking algorithms which perform better in terms of throughput than their lock-free or wait-free counterparts[5]. Still, non-blocking algorithms can shine in high-contention settings. A small number of CAS operations can amount to less overhead than aquiring a lock, and more fine-grained concurrency coupled with progress guarantees *can* reduce contention[6].

Using Fetch-and-Add to Reduce Contention

The atomic Fetch-and-Add (F&A) instruction adds a value to an integer and returns the old or new value of that integer. Here are the basic semantics of the operation in Go[7]:

```
//atomically
func AtomicFetchAdd(src *int, delta int) {
  *src += delta
  return *src
}
```

[5] Consult the related work sections of Yang and Mellor-Crummey (2016) on *combining* queues for an example of this; Morrison and Afek (2013) has a similar survey

[6] Less contention is not something that you get automatically when the algorithm is lock-free. An early lock-free queue Michael and Scott (1996) still suffers from from bottlenecks around the head and tail pointers all being CAS-ed by contending threads. Most of these CASes will fail, and all threads whose CASes fail must retry. Exponential backoff schemes can help this state of affairs but the bottleneck is still present; see the performance measurements in Yang and Mellor-Crummey (2016) with includes the algorithm from Michael and Scott (1996).

[7] F&A is more commonly defined to return the *old* value of `src`, but returning the new value is equivalent.

While hardware support for a F&A instruction is not as universal as that of CAS, F&A is implemented on x86. On modern x86 machines, F&A is much faster than CAS (see Morrison and Afek 2013 for performance measurements), and it always succeeds. This has the dual effect allowing code making judicious use of F&A to be both efficient and easier to reason about than equivalents that rely only on CAS. A common pattern exemplifying this idea is to first use F&A to acquire an index into an array, and then to use more conventional techniques to write to that index. This is helpful because it can reduce contention on individual locations for a data-structure.

A Non-blocking Queue From an Infinite Array

To illustrate this, we will write two non-blocking queues in pseudo-Go based on an infinite array (`Queue2` is based on the obstruction-free queue presented in pseudo-code in Yang and Mellor-Crummey 2016, `Queue1` is a CAS-ification of that design). Both of these designs make use of the fact that head and tail pointers *only ever increase*.

```
type Queue1 struct {
    head, tail *T
    data       [ω]T
}
func (q *Queue1) Enqueue(elt T) {
    for {
        newTail := atomic.LoadPointer(&q.tail) + 1
        if atomic.CompareAndSwapT(newTail, nil, elt) {
            atomic.CompareAndSwap(&q.tail, q.tail, newTail)
            break
        }
    }
}
func (q *Queue1) Dequeue() T {
    for {
        curHead := atomic.LoadPointer(&q.head)
        curTail := atomic.LoadPointer(&q.tail)
        if curHead == curTail {
            return nil
        }
        if atomic.CompareAndSwapPointer(&q.head, curHead, curHead+1) {
            return *curHead
        }
    }
}
```

The second queue will assume that the type `T` can not only take on a `nil` value but also an unambiguous `SENTINEL` value that a user is guaranteed not to pass in to `Enqueue`. This value is used to mark an index as unusable, signalling a conflicting `Enqueue` thread that it should try again.

```
type Queue2 struct {
    head, ta uint
```

```
28      data     [ω]T
29  }
30
31  func (q *Queue2) Enqueue(elt T) {
32      for {
33          myTail := atomic.AddUint(&q.tail) - 1
34          if atomic.CompareAndSwapT(&q.data[myTail], nil, elt) {
35              break
36          }
37      }
38  }
39
40  func (q *Queue2) Dequeue() T {
41      for {
42          myHead := atomic.AddUint(&q.head) - 1
43          curTail := atomic.LoadUint(&q.tail)
44          if !atomic.CompareAndSwapPointer(&q.data[myHead], nil, SENTINEL) {
45              return atomic.LoadT(&q.data[myHead])
46          }
47          if myHead == curTail {
48              return nil
49          }
50      }
51  }
```

The core algorithm for both `Queue1` and `Queue2` is essentially the same. Enqueueing threads load a view of the tail pointer and try to CAS their element in one element after that pointer; dequeueing threads perform a symmetric operation to advance the head pointer. The practical (that is, practical for algorithms that require a infinite amount of memory) difference between `Queue1` and `Queue2` is that `Queue2` first has threads perform an atomic increment of a head or tail index. This means that two concurrent enqueue operations will always attempt a CAS on *different* queue elements. As a result, enqueue operations need only concern themselves with dequeue operations that increment `head` to the same value as their `myTail` (lines 33–34).

A downside of this approach is that while `Queue1` is lock free, `Queue2` is merely obstruction free. For an enqueue/dequeue pair of threads, each can continually increment equal `head` and `tail` indices while the dequeuer's CAS (line 44) always succeeds before the enqueuer's (line 34) resulting in livelock.

Lessons for Channels

The `Queue2` above is the core of the implementation of a fast wait-free queue in Yang and Mellor-Crummey (2016). It is also the basic idea that we will leverage when designing a more scalable channel. The rest of their algorithm consists in solving three problems that have analogs in our setting.

(1) *Simulating an infinite array with a finite amount of memory.* Here the authors implement

a linked list of fixed-length arrays (called segments, or cells); threads grow this array when more space is required.

(2) *Going from obstruction freedom to wait freedom.* This involves attempting either `Dequeue` or `Enqueue` above for a constant number of iterations, followed by a slow path which involves implementing a helping mechanism[8] to help contending threads to finish their outstanding operations.

(3) *Memory Reclamation.* Reclaiming memory in a non-blocking setting is, perhaps unsurprisingly, a very fraught task.

While the solution to (3) in this paper is interesting and efficient, we will (mercifully) be relying on Go's garbage collection mechanism to solve this problem. For (1) we will employ essentially the same algorithm as the paper, but with additional optimizations for memory allocation. For (2) our slow path will implement the blocking semantics of a channel.

An Unbounded Channel With Low Contention

We first consider the case of implementing an unbounded channel. While this channel is blocking — Go channels must in some capacity be blocking as they provide a synchronization mechanism — it only blocks when it has to (i.e. for receives that do not yet have a corresponding send), and when it does progress is impeded for at most 2 threads, the components of a send/receive pair. We will start with the types:

```
type Elt *interface{}
type index uint64

// segment size
const segShift = 12
const segSize = 1 << segShift

// The channel buffer is stored as a linked list of fixed-size arrays of size
// segsize. ID is a monotonically increasing identifier corresponding to the
// index in the buffer of the first element of the segment, divided by segSize
// (see SplitInd).
type segment struct {
    ID   index // index of Data[0] / segSize
    Next *segment
    Data [segSize]Elt
}

// Load atomically loads the element at index i of s
func (s *segment) Load(i index) Elt {
```

[8] Helping is a standard technique for making obstruction-free or lock-free algorithms wait free. The technique goes back to M. Herlihy (1991); the practice of using a weaker progress guarantee as a fast path and then falling back to a helping mechanism to ensure wait freedom was introduced in Kogan and Petrank (2012). An explanation of helping can be found in M. Herlihy and Shavit (2008) chapters 6, 10.5.

```
    return Elt(atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&s.Data[i]))))
}

// Queue is the global state of the channel. It contains indices into the head
// and tail of the channel as well as a linked list of spare segments used to
// avoid excess allocations.
type queue struct{ H, T index }

// Thread-local state for interacting with an unbounded channel
type UnboundedChan struct {
    // pointer to global state
    q *queue
    // pointer into last guess at the true head and tail segments
    head, tail *segment
}
```

The only data-structure-global global state that we employ is the `queue` structure which maintains the head and tail indices. Pointers into the data itself are kept locally in an `UnboundedChan` for two reasons

(1) It reduces any possible contention resulting from updated shared head or tail pointers.

(2) If individual threads all update local head and tail pointers, then the garbage collector will be able to clean up used segments when (and only when) all threads no longer hold a reference to them.

We note that a downside of this design is that inactive threads that hold such a handle can cause space leaks by holding onto references to long-dead segments.

Users interact with a channel by first creating an initial value, and later cloning that value and others derived from it using `NewHandle`.

```
// New initializes a new queue and returns an initial handle to that queue. All
// other handles are allocated by calls to NewHandle()
func New() *UnboundedChan {
    segPtr := &segment{} // 0 values are fine here
    q := &queue{H: 0, T: 0}
    h := &UnboundedChan{q: q, head: segPtr, tail: segPtr}
    return h
}

// NewHandle creates a new handle for the given Channel
func (u *UnboundedChan) NewHandle() *UnboundedChan {
    return &UnboundedChan{q: u.q, head: u.head, tail: u.tail}
}
```

Sending and Receiving

The key enqueue (or send) algorithm is to atomically increment the `T` index, attempt to CAS in the item, and to wake up a blocking thread if the CAS fails. We will begin with the `Enqueue` code and then explain the code that it calls.

```
1  func (u *UnboundedChan) Enqueue(e Elt) {
2      u.adjust()
3      myInd := index(atomic.AddUint64((*uint64)(&u.q.T), 1) - 1)
4      cell, cellInd := myInd.SplitInd()
5      seg := u.q.findCell(u.tail, cell)
6      if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&seg.Data[cellInd])),
7          unsafe.Pointer(nil), unsafe.Pointer(e)) {
8          return
9      }
10     wt := (*waiter)(atomic.LoadPointer(
11         (*unsafe.Pointer)(unsafe.Pointer(&seg.Data[cellInd]))))
12     wt.Send(e)
13 }
14
15 func (u *UnboundedChan) Dequeue() Elt {
16     u.adjust()
17     myInd := index(atomic.AddUint64((*uint64)(&u.q.H), 1) - 1)
18     cell, cellInd := myInd.SplitInd()
19     seg := u.q.findCell(u.head, cell)
20     elt := seg.Load(cellInd)
21     wt := makeWaiter()
22     if elt == nil &&
23         atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&seg.Data[cellInd])),
24             unsafe.Pointer(nil), unsafe.Pointer(wt)) {
25         return wt.Recv()
26     }
27     return seg.Load(cellInd)
28 }
```

The `adjust` (line 2) method atomically loads `H` and `T`, then advances `u.head` and `u.tail` to point to their cells. The `AtomicAdd` on line 3 acquires an index into the queue. `SplitInd` (line 4) returns the cell ID and the index into that cell corresponding to `myInd`. As `T` can only increase, the only possible thread that could also be contending for this item is a `Dequeue`ing thread that acquired `H` as the same value as `myInd`. So it comes down to the CASes on lines 6 and 21–22. If the first CAS fails, it means a `Dequeue` thread has swapped in a `waiter`, if it succeeds then it means an `Enqueuer` can return and a contending `Dequeuer` can just load the value in `cellInd`.

Blocking

So what is a `waiter`? It acts like a channel with buffer size 1, or an *MVar* in the Haskell community (see Chapter 7 of Marlow 2012 for an introduction), that can only tolerate 1 element being sent on

it. We currently implement this in terms of a single value and a `WaitGroup`. `WaitGroups` in Go's `sync` package allow goroutines to `Add` an integer value to the `WaitGroup`'s counter and to `Wait` for that counter to reach zero. If the counter goes below zero, the current `WaitGroup` implementation panics, which is helpful for debugging purposes as there should only ever be one `Send` or `Recv` on a `waiter` here.

```
type waiter struct {
    E      Elt
    Wgroup sync.WaitGroup
}
```

```
func makeWaiter() *waiter {
    wait := &waiter{}
    wait.Wgroup.Add(1)
    return wait
}
```

```
func (w *waiter) Send(e Elt) {
    atomic.StorePointer((*unsafe.Pointer)(unsafe.Pointer(&w.E)), unsafe.Pointer(e))
    w.Wgroup.Done() // The Done method just calls Add(-1)
}

func (w *waiter) Recv() Elt {
    w.Wgroup.Wait()
    return Elt(atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&w.E))))
}
```

There are two important parts of our strategy to implement blocking. Neither Enqueuers nor Dequeuers will block at all if Enqueuers complete before Dequeuers begin. In fact the only global synchronization they must perform is a single F&A and a single *uncontended* CAS (unless they must grow the queue; see below). Second, if a Enqueuer does not arrive soon enough and must block on a `waiter`, there will be essentially no contention for the waiter because there can only be one other threads interacting with it.

Growing the Queue and Allocation

We will now describe the implementation of the `findCell` method. The algorithm is to start at a given segment pointer, and to follow that segment's `Next` pointer until that segment's `ID` is equal to a given `cell` index. If `findCell` reaches the end of the list of segments before it reaches the correct index, it attempts to allocate a new segment and place it onto the end of the list. Here is some code:

```
1 func (q *queue) findCell(start *segment, cellID index) *segment {
2     cur := start
3     for cur.ID != cellID {
4         next := (*segment)(atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&cur.Next))))
5         if next == nil {
6             q.Grow(cur)
7             continue
8         }
9         cur = next
```

```
    }
    return cur
}
func (q *queue) Grow(tail *segment) {
    curTail := atomic.LoadUint64((*uint64)(&tail.ID))
    newSegment := &segment{ID: index(curTail + 1)}
    if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&tail.Next)),
        unsafe.Pointer(nil), unsafe.Pointer(newSegment)) {
        return
    }
}
```

Note that we can get away with performing a single CAS operation in `Grow` because if our CAS failed we know someone else succeeded, and a new segment with ID of `tail.ID+1` is the only possible value that could be placed there. However, there *is* a problem with this implementation: it is extremely wasteful. In a high-contention situation, it is possible for many threads to all allocate a new segment, but only one of those threads will succeed. Any failed allocations will become immediately unreachable and will hence be garbage collected. In our experiments, channel operations are fastest when segments have a size of ≥ 1024, so any wasted allocation can have a tangible impact on throughput. This slowdown was evident in our performance measurements.

Our solution to this problem is to keep a lock-free linked list of segments in the `queue` structure. Threads in `Grow` first try and pop a segment off of this list, and then perform the CAS. Only if this pop fails do they allocate a new segment. Symmetrically, if a CAS fails then threads attempt to push a segment onto this list. The list keeps a best-effort counter representing its length and does not allow this counter to grow past a maximum length; this allows us to avoid a space leak in the implementation of the queue. For a full implementation of `Grow`, see Appendix A.

Extending to the Bounded Case

Go channels do not have an unbounded variant. While the structure offered above is potentially useful, there are good reasons to prefer bounded channels in some settings[9]. Unbuffered channels allow for a more synchronous programming model that is common in Go to synchronize two cooperating threads; this level of synchronization is useful to have. This section describes the implementation of a bounded channel on top of the unbounded implementation above.

Preliminaries

We re-use the `q` and `segment` types, along with the `findCell` and `Grow` machinery. Almost all of the difference is in the new `Enqueue` and `Dequeue` operations. These are, however, significantly more complex. This complexity is the result of senders and receivers being given new responsibilities:

- Senders must decide if they should block and wait for more receivers to arrive.

[9] See, for example, this discussion on the Rust mailing list regarding unbounded channels. Haskell's standard channel implementation in Control.Concurrent is unbounded, as are the STM variants.

- Receivers have to wake up any waiters who ought to wake up if they succeed in popping an element off of the queue.

As before, this protocol is implemented in a manner that avoids blocking unless blocking is required by the channel semantics. This means `Enqueue` and `Dequeue` methods must consider arbitrary interleavings of the unbounded channel protocol and the new blocking protocol. The `BoundedChan` has an additional integer field `bound` indicating the maximum number of senders permitted to return without having rendezvoused with a receiver.

We also introduce an immutable global `sentinel` pointer used by receiving threads to signal that a sender should not block. A consequence of this design is that now all places that required a CAS from `nil` to another value must also attempt to CAS from `sentinel`. We maintain the invariant that no value will transition from `sentinel` back to `nil`, so the `tryCas` function below guarantees that `seg.Data[segInd]` is neither `nil` nor `sentinel` when it returns (unless `e` is either of those).

(Aside) Possible Histories of an Element in a Segment

In the unbounded case, there were essentially two possible histories of a value in the queue:

Events	History
Sender, Receiver	`nil` → `Elt`
Receiver, Sender	`nil` → `*waiter`

This can be viewed as the key invariant that is enforced in the implementation of unbounded channels. There are more histories in the bounded case. These (and only these) can all arise — keeping this in mind is helpful for understanding the protocol:

Events	History
Sender, Receiver	`nil` → `Elt`
Receiver, Sender	`nil` → `*waiter`
Waker, Sender, Receiver	`nil` → `sentinel` → `Elt`
Waker, Receiver, Sender	`nil` → `sentinel` → `*waiter`
Sender†, Waker, Sender, Receiver	`nil` → `*weakWaiter` → `Elt`
Sender†, Waker, Receiver, Sender	`nil` → `*weakWaiter` → `*waiter`

Where Sender† denotes that a sender arrives but must block for more receivers to complete, and a Waker is any thread that successfully wakes up a blocked Sender. The details of what a `weakWaiter` is and who exactly plays the role of "Waker" are covered in the following sections.

Enqueue

We first present the source of `tryCas` and `Enqueue`:

```
func tryCas(seg *segment, segInd index, elt unsafe.Pointer) bool {
    return atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&seg.Data[segInd])),
        unsafe.Pointer(nil), elt) ||
        atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&seg.Data[segInd])),
            sentinel, elt)
}

// Enqueue sends e on b. If there are already >=bound goroutines blocking, then
// Enqueue will block until sufficiently many elements have been received.
func (b *BoundedChan) Enqueue(e Elt) {
    b.adjust()
    startHead := index(atomic.LoadUint64((*uint64)(&b.q.H)))
    myInd := index(atomic.AddUint64((*uint64)(&b.q.T), 1) - 1)
    cell, cellInd := myInd.SplitInd()
    seg := b.q.findCell(b.tail, cell)
    if myInd > startHead && (myInd-startHead) > index(uint64(b.bound)) {
        // there is a chance that we have to block
        var w interface{} = makeWeakWaiter(2)
        if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&seg.Data[cellInd])),
            unsafe.Pointer(nil), unsafe.Pointer(Elt(&w))) {
            // we successfully swapped in w. No one will overwrite this
            // location unless they send on w first. We block.
            w.(*weakWaiter).Wait()
            if atomic.CompareAndSwapPointer(
                (*unsafe.Pointer)(unsafe.Pointer(&seg.Data[cellInd])),
                unsafe.Pointer(Elt(&w)), unsafe.Pointer(e)) {
                return
            } // someone put a waiter into this location. We need to use the slow path
        } else if atomic.CompareAndSwapPointer(
            (*unsafe.Pointer)(unsafe.Pointer(&seg.Data[cellInd])),
            sentinel, unsafe.Pointer(e)) {
            // Between us reading startHead and now, there were enough
            // increments to make it the case that we should no longer
            // block.
            return
        }
    } else {
        // normal case. We know we don't have to block because b.q.H can only
        // increase.
        if tryCas(seg, cellInd, unsafe.Pointer(e)) {
            return
        }
    }
    ptr := atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&seg.Data[cellInd])))
```

```
45      w := (*waiter)(ptr)
46      w.Send(e)
47      return
48  }
```

`Enqueue` starts by loading a value of `H` and then acquiring `myInd`. Note that this *is not* a consistent snapshot of the state of the queue, as `H` could have moved between loading it and incrementing `myInd` (lines 12–13). However, `H` will only increase! If `startHead` is within `b.bound` of `myInd` it means that `H` is at most that far behind `T` was when we performed the increment. In that case we can simply attempt to CAS in `e` (line 40). If that fails, it can only mean that a receiver has placed a `waiter` in this index, so we wake up the receiver and return (lines 44–46).

If there is a chance that we *do* have to block, then we allocate a new `weakWaiter`. A `weakWaiter` is like a `waiter` except it does not contain a value, but it does allow for more than one message to be received. There are many ways to implement such a construct in Go, here is an implementation in terms of a `WaitGroup`:

```
type weakWaiter struct {
    OSize, Size  int32
    Wgroup sync.WaitGroup
}
func makeWeakWaiter(i int32) *weakWaiter {
    wait := &weakWaiter{Size: i, OSize: i}
    wait.Wgroup.Add(1)
    return wait
}
func (w *weakWaiter) Signal() {
    newVal := atomic.AddInt32(&w.Size, -1)
    orig := atomic.LoadInt32(&w.OSize)
    if newVal+1 == orig { w.Wgroup.Done() }
}
```

In the that case we may block, we construct a `weakWaiter` with a buffer size of two because it is possible to have two dequeueing threads concurrently attempt to wake up an enqueueing thread (see below). If the sender successfully CASes `w` into the proper location (line 19), then it waits and attempts the rest of the unbounded channel protocol when it wakes. There are two possible scenarios if this CAS fails:

(1) A receiver for `b.bound` elements forward in the channel attempted to wake up this sender, but arrived before `w` was stored.
(2) A receiver has already started waiting at this location

The CAS on line 29 determines which case this is. If (1) then the CAS will fail and the sender must now wake up the waiting receiver thread on line 46. If (2) is the case then the CAS will succeed and `e` will successfully be in the queue.

Dequeue

The `Dequeue` implementation effectively mirrors the `Enqueue` implementation. There are, however, a few things that are especially subtle. Let's start with the implementation:

```
49  func (b *BoundedChan) Dequeue() Elt {
50      b.adjust()
51      myInd := index(atomic.AddUint64((*uint64)(&b.q.H), 1) - 1)
52      cell, segInd := myInd.SplitInd()
53      seg := b.q.findCell(b.head, cell)
54      // If there are Enqueuers waiting to complete due to the buffer size, we
55      // take responsibility for waking up the thread that FA'ed b.q.H + b.bound.
56      // If bound is zero, that is just the current thread. Otherwise we have to
57      // do some extra work. The thread we are waking up is referred to in names
58      // and comments as our 'buddy'.
59      var (
60          bCell, bInd index
61          bSeg        *segment
62      )
63      if b.bound > 0 {
64          buddy := myInd + index(b.bound)
65          bCell, bInd = buddy.SplitInd()
66          bSeg = b.q.findCell(b.head, bCell)
67      }
68      w := makeWaiter()
69      var res Elt
70      if tryCas(seg, segInd, unsafe.Pointer(w)) {
71          res = w.Recv()
72      } else {
73          // tryCas failed, which means that through the "possible histories"
74          // argument, this must be either an Elt, a waiter or a weakWaiter. It
75          // cannot be a waiter because we are the only actor allowed to swap
76          // one into this location. Thus it must either be a weakWaiter or an Elt.
77          // if it is a weakWaiter, then we must send on it before casing in w,
78          // otherwise the other thread could starve. If it is a normal Elt we
79          // do the rest of the protocol. This also means that we can safely load
80          // an Elt from seg, which is not always the case because sentinel is
81          // not an Elt.
82          // Step 1: We failed to put our waiter into Ind. That means that either our
83          // value is in there, or there is a weakWaiter in there. Either way these
84          // are valid elts and we can reliably distinguish them with a type assertion
85          elt := seg.Load(segInd)
86          res = elt
87          if ww, ok := (*elt).(*weakWaiter); ok {
88              ww.Signal()
89              if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&seg.Data[segInd]))
90                  unsafe.Pointer(elt), unsafe.Pointer(w)) {
91                  res = w.Recv()
```

```
92              } else {
93                  // someone cas'ed a value from a weakWaiter, could only have been our
94                  // friend on the dequeue side
95                  res = seg.Load(segInd)
96              }
97          }
98      }
99      for b.bound > 0 { // runs at most twice
100         // We have successfully gotten the value out of our cell. Now we
101         // must ensure that our buddy is either woken up if they are
102         // waiting, or that they will know not to sleep.
103         // if bElt is not nil, it either has an Elt in it or a weakWater. If
104         // it has a waitch then we need to send on it to wake up the buddy.
105         // If it is not nill then we attempt to cas sentinel into the buddy
106         // index. If we fail then the buddy may have cas'ed in a wait
107         // channel so we must go again. However that will only happen once.
108         bElt := bSeg.Load(bInd)
109         if bElt != nil {
110             if ww, ok := (*bElt).(*weakWaiter); ok {
111                 ww.Signal()
112             }
113             // there is a real queue value in bSeg.Data[bInd], therefore
114             // buddy cannot be waiting.
115             break
116         }
117         // Let buddy know that they do not have to block
118         if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&bSeg.Data[bInd])),
119             unsafe.Pointer(nil), sentinel) {
120             break
121         }
122     }
123     return res
124 }
```

Now the subtleties. A dequeuer may have to wake up multiple waiting send threads: the one waiting at `myInd` and the other waiting at `myInd + bound` (or `bInd`). This may seem strange because the dequeuer that receives `myInd-bound` ought to have woken up any pending senders. The issue is that *we have no gaurantee that this dequeuer has returned*. The possibility of this occurring is remote with a large buffer size, but when `bound` is small it happens with some regularity.

The second is a peculiarity of Go. On line 87 there is a *type assertion* which de-references an `Elt` to yield a value of type `interface{}`. The `interface{}` contains a pointer to some runtime information about the actual type of the pointed-to struct, and the `.(*weakWaiter)` syntax queries if `elt` is a pointer to a weakWaiter. This is a safe thing to do because `weakWaiter` is a package-private type: no external caller could pass in an `Elt` that pointed to a `weakWaiter` unless we returned one from any of the public functions in the package, which we do not.

This is complicated by the fact that `*waiters` are actually stored in the queue directly, without hiding behind an interface value (e.g. at line 90). This is because the extra layer of indirection is

unnecessary: it is always possible to determine where an `Elt` or a `*waiter` is present in a given location based on which CAS's have failed and which have succeeded.

Performance

We benchmarked 5 separate channels on enqueue/dequeue pairs:

- *Bounded0*: A `BoundedChan` with buffer size 0
- *Bounded1K*: A `BoundedChan` with buffer size 1024
- *Unbounded*: An `UnboundedChan`
- *Chan0*: An unbuffered native Go channel
- *Chan1K*: A native Go channel with buffer size 1024
- *Chan10M*: A native Go channel with buffer size 10^7, which is the total number of elements enqueued into the channel over the course of the benchmark.

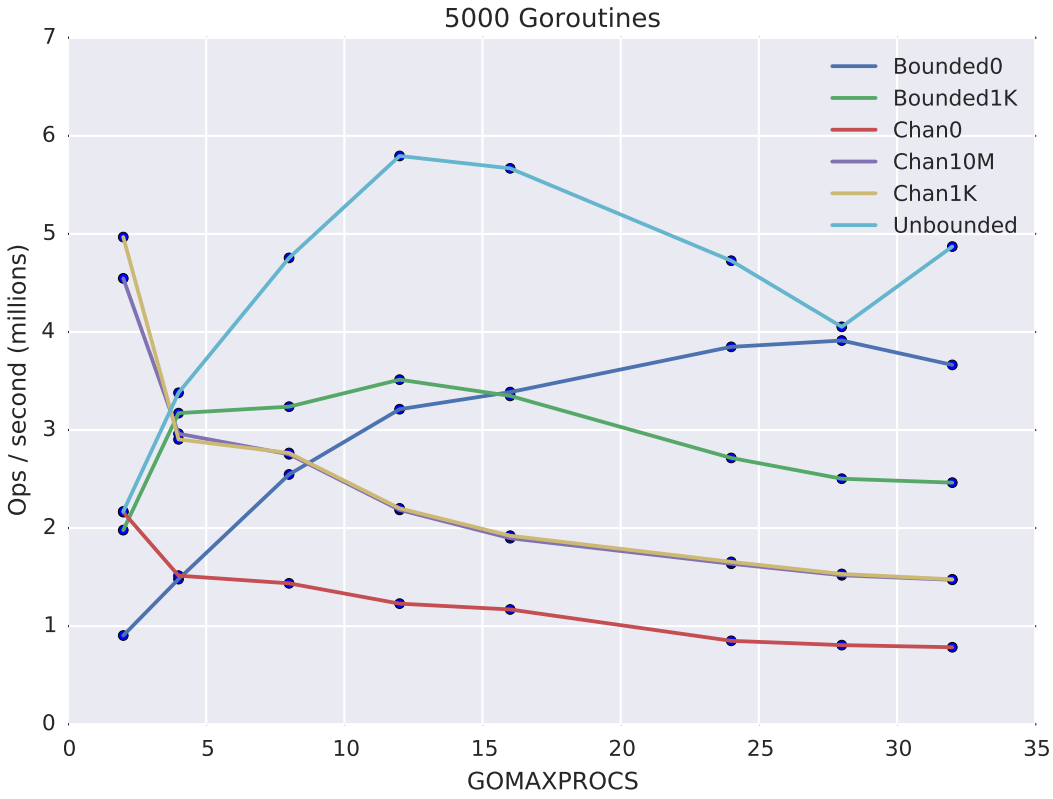
We include benchmark results for two cases: one where we allocate one goroutine per processor (where processors are set with the `GOMAXPROCS` procedure from Go's runtime), and one where we allocate 5000 goroutines, irrespective of the current value of `GOMAXPROCS`. We include both of these for two reasons. First, it is not uncommon to have thousands of goroutines active in a running Go program and it makes sense to consider the case where processors are over-subscribed in that manner. Second, we noticed that performance is often *better* in the cases where cores are oversubscribed. While counter-intuitive, this is possibly due to a combination of unpredictable scheduler performance, and the lower overhead of synchronizing between two goroutines executing on the same core.

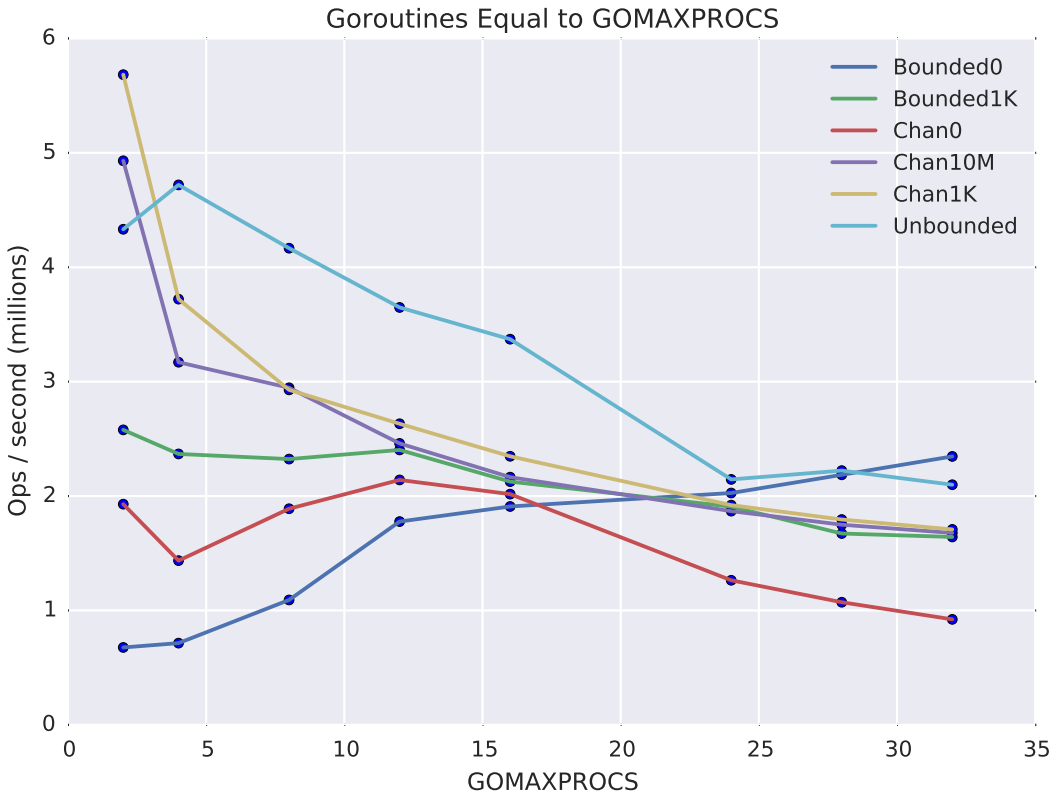
These benchmarks were conducted on a machine with 2 Intel Xeon 2620 v4 CPUs each with 8 cores clocked at 2.1GHz with two hardware threads per core. We were unable to allocate cores in an intuitive manner, so the 16-core benchmark is actually using all of a single CPU's hardware threads; only at core-counts higher than 16 does the program cross a NUMA-domain. The benchmarks were run on the Windows Subsystem for Linux[10]; an implementation of an Ubuntu 14.04 userland from within the Windows 10 Operating System. These benchmarks were conducted using Go version 1.6.

These numbers were produced by performing 5,000,000 enqueues and dequeues per configuration, averaged over 5 iterations per setting, with a full GC between iterations.

The benchmarks show that both *Bounded* and *Unbounded* are able to increase throughput as the core-count increases. Native Go channels are unable to do so. When using more than 4 processors, *Unbounded* and *Bounded1K* provide much better throughput than native channels regardless of buffer size. *Unbounded* in particular is often 2-3x faster than the buffered *Chan* configurations, while *Bounded0* continues to increase throughput even after crossing a NUMA domain and dipping into using multiple hardware threads per core. At the highest core counts, all three new configurations outpace native Go channels.

[10] See this blog post as well as the various follow-ups for an overview of this system.


5000 Goroutines
Ops / second (millions)
GOMAXPROCS
Bounded0
Bounded1K
Chan0
Chan10M
Chan1K
Unbounded
0
1
2
3
4
5
6
7
0
5
10
15
20
25
30
35


Goroutines Equal to GOMAXPROCS
Ops / second (millions)
GOMAXPROCS
Bounded0
Bounded1K
Chan0
Chan10M
Chan1K
Unbounded
0
1
2
3
4
5
6
0
5
10
15
20
25
30
35

Linearizability

We contend that both the bounded and unbounded queues presented in this document are *linearizable* with respect to their Enqueue and Dequeue operations. Linearizability is a strong consistency guarantee often used to specify the behavior of concurrent data-structures. Informally we say a structure is linearizable if for an arbitrary (possibly infinite) history of concurrent operations on the structure beginning and ending at specific times, we can *linearize* it such that each each operation occurs atomically at some point in time between it beginning and ending (See Chapter 3 of M. Herlihy and Shavit 2008 for an overview; Linearizability was introduced with M. P. Herlihy and Wing 1990).

This section describes linearization procedures for the bounded and unbounded channels in this document. Both channels begin with a fetch-add on the head or tail index for the queue that determines the *logical index* that will be the subject of their send or receive. We denote e_i and d_i the enqueue and dequeue operations that fetch-add to get a value of `myInd` equal to i. We will provide linearizations that preserve the following properties, where $\prec$ indicates precedence in the linearized sequence of events. For all i we must have that

(1) $e_i \prec e_{i+1}$ (if e_{i+1} occurs)
(2) $d_i \prec d_{i+1}$ (if d_{i+1} occurs)
(3) $e_i \prec d_i$ (if both occur)

Which we take to be a straight-forward sequential specification for a channel.

Unbounded Channels

Our linearization procedure considers two broad cases, a fast and slow path.

- In the fast path there is sufficient distance between enqueuers and dequeuers such that the fetch-add of e_i occurs before the fetch-add for d_i *and* e_i's CAS succeeds. In this case, linearize e_i and d_i at their respective fetch-adds.
- In the case where d_i's fetch-add occurs before that of e_i (or the CAS fails) we linearize *both* operations at e_i's fetch-add, with e_i occurring just before d_i.

Observe that both cases in this procedure linearize e_i, d_i between them starting and finishing. The second case is guaranteed to do so because if d_i must block then e_i is responsible for unblocking them, and if d_i does not block then we know its CAS fails, meaning that e_i's fetch-add occurs after d_i's fetch-add but before its failed CAS.

We will now show that the above procedure yields a history consistent with the three criteria provided above. The proof strategy is to show, for both the fast and slow paths, that we can maintain the criteria for an arbitrary $e_i, d_i, e_{i+1}, d_{i+1}$. Given this we can conclude that the criteria are satisfied for an arbitrary number of enqueue-dequeue pairs. We then consider the other possible cases.

The Fast Path

We know that we satisfy (1) because all e_i, fast or slow path, linearize at their fetch-add, and these are guaranteed to provide a total ordering on operations. We satisfy (3) by assumption. Consider d_{i+1}, if it hits the fast path then it is linearized at its fetch-add which must happen after d_i's fetch-add. If it hits the slow path then it will be linearized at the fetch-add of e_{i+1}, but by assumption we only hit the slow path if d_{i+1}'s fetch-add completed before that of e_{i+1}; d_{i+1}'s fetch-add definitely completed after that of d_i, so we satisfy (2).

The Slow Path

The argument for (1) is the same as in the fast path, and the argument for (3) follows by assumption. Once again, the interesting case is to show that we maintain an ordering between dequeue operations. There are two possible cases:

d_{i+1} blocks

: We know that d_{i+1} will take the slow path, and will therefore be linearized at a later fetch-add.

d_{i+1} does not block

: The only way that d_{i+1} does not block is if its CAS fails, which means that there is another enqueuer e_{i+1} that completed. Regardless of whether d_{i+1} is linearized on a slow path or a fast path, it must be after the fetch-add in e_{i+1} and hence also that of e_i.

Small Numbers of Operations

If there is only one enqueue operation, then at most one dequeue operation will be linearized. This is fine, because at most one dequeue operation will complete, while any others will block forever. The definitions of the two cases in the linearization procedure automatically yield condition (3), while (1,2) are trivially satisfied as there is only one enqueue and at most one dequeue.

Concluding

We can conclude by induction that for any finite number of enqueues and dequeues, there is a linearization that satisfies a standard sequential specification for a channel. For infinite sequences of operations (assuming `H` and `T` can be updated with with arbitrary precision) there is probably a similar co-inductive characterization of the same process; the above argument should still hold. We conclude that unbounded channels are linearizable. □

Bounded Channels

The bounded case has the same linearization procedure (and proofs) as the unbounded case, with the caveat that enqueue operations that do not return never make it into the history. This works because all operations unconditionally perform fetch-adds, even if they later have to block for an unbounded amount of time. □

Conclusion and Future Work

This document demonstrates that it is possible to have scalable unbounded and bounded queues while still satisfying a strong consistency guarantee. It leverages techniques from the recent literature on non-blocking queues to implement (to our knowledge) novel blocking constructs. There are a number of avenues for future work.

Verification

: It will be useful to model both channels in SPIN or TLA+ to provide further assurance that the algorithms are correct. While it would be more involved, proving correctness in Coq in line with techniques mentioned in FRAP would also be helpful in building confidence in the algorithms.

Implement in the Go runtime

: Implementing these channels within the runtime could further reduce these algorithms' overhead. In particular they will allow for more efficient implementation of the blocking semantics in that they can access goroutine and scheduling metadata directly, whereas the current implementation relies on `WaitGroups`, which may be too heavyweight for our purposes.

Improving Performance

: Some variants of this algorithm still perform worse at lower core-counts than their native Go equivalents. One possible reason for this is how much allocation these queues perform (go channels need only keep a single fix-sized buffer). It could be fruitful to experiment with schemes that reduce allocation, as well as algorithms that allocate a fix-sized buffer, similar to the CRQ algorithm in Morrison and Afek (2013).

Appendix A: Efficient Segment Allocation

In order to speed up allocation, we add a list to the queue state. This list is similar to standard lock-free queue designs in the literature, and bares some resemblance to `Queue1` above. The major difference here is that we only provide partial push and pop operations: Push will fail if the list may be too large or if it runs out of `patience`, and Pop will fail if its CAS fails more than `patience` times.

```
type listElt *segment
type segList struct {
    MaxSpares, Length int64
    Head              *segLink
}

type segLink struct {
    Elt  listElt
    Next *segLink
}

func (s *segList) TryPush(e listElt) {
    // bail out if list is at capacity
    if atomic.LoadInt64(&s.Length) >= s.MaxSpares {
        return
    }
    // add to length. Note that this is not atomic with respect to the append,
    // which means we may be under capacity on occasion. This list is only used
    // in a best-effort capacity, so that is okay.
    atomic.AddInt64(&s.Length, 1)
    tl := &segLink{Elt: e, Next: nil}
    const patience = 4
    i := 0
    for ; i < patience; i++ {
        // attempt to cas Head from nil to tail,
```

```
        if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&s.Head)),
            unsafe.Pointer(nil), unsafe.Pointer(tl)) {
            break
        }
        // try to find an empty element
        tailPtr := (*segLink)(atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&s.Head))))
        if tailPtr == nil {
            // if Head was switched to nil, retry
            continue
        }
        // advance tailPtr until it has anil next pointer
        for {
            next := (*segLink)(atomic.LoadPointer(
                (*unsafe.Pointer)(unsafe.Pointer(&tailPtr.Next))))
            if next == nil {
                break
            }
            tailPtr = next
        }
        // try and add something to the end of the list
        if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&tailPtr.Next)),
            unsafe.Pointer(nil), unsafe.Pointer(tl)) {
            break
        }
    }
    if i == patience {
        atomic.AddInt64(&s.Length, -1)
    }
}

func (s *segList) TryPop() (e listElt, ok bool) {
    const patience = 1
    if atomic.LoadInt64(&s.Length) <= 0 {
        return nil, false
    }
    for i := 0; i < patience; i++ {
        hd := (*segLink)(atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&s.Head))))
        if hd == nil {
            return nil, false
        }
        // if head is not nil, try to swap it for its next pointer
        nxt := (*segLink)(atomic.LoadPointer((*unsafe.Pointer)(unsafe.Pointer(&hd.Next))))
        if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&s.Head)),
            unsafe.Pointer(hd), unsafe.Pointer(nxt)) {
            atomic.AddInt64(&s.Length, -1)
            return hd.Elt, true
        }
    }
```

```
    return nil, false
}
```

Given this list implementation, we simply insert calls to `TryPush` and `TryPop` around the original implementation of `Grow` to have it take advantage of extra allocations:

```
type queue struct {
    H, T        index
    SpareAllocs segList
}
func (q *queue) Grow(tail *segment) {
    curTail := atomic.LoadUint64((*uint64)(&tail.ID))
    if next, ok := q.SpareAllocs.TryPop(); ok {
        atomic.StoreUint64((*uint64)(&next.ID), curTail+1)
        if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&tail.Next)),
            unsafe.Pointer(nil), unsafe.Pointer(next)) {
            return
        }
    }
    newSegment := &segment{ID: index(curTail + 1)}
    if atomic.CompareAndSwapPointer((*unsafe.Pointer)(unsafe.Pointer(&tail.Next)),
        unsafe.Pointer(nil), unsafe.Pointer(newSegment)) {
        return
    }
    // If we allocated a new segment but failed, attempt to place it in
    // SpareAlloc so someone else can use it.
    q.SpareAllocs.TryPush(newSegment)
}
```

This scheme led to significant speedups in performance tests, but the code in `q.go` includes a constant that, if set to false, will disable any such list-based caching of allocations. This should make it easy to verify or falsify those performance measurements.

References

Donovan, Alan A.A., and Brian W. Kernighan. 2015. *The Go Programming Language*. 1st ed. Addison-Wesley Professional.

Herlihy, Maurice. 1991. "Wait-Free Synchronization." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13 (1): 124–149.

Herlihy, Maurice P, and Jeannette M Wing. 1990. "Linearizability: a Correctness Condition for Concurrent Objects." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (3): 463–492.

Herlihy, Maurice, and Nir Shavit. 2008. "The Art of Multiprocessor Programming."

Hoare, Charles Antony Richard. 1978. "Communicating Sequential Processes." In *The Origin of Concurrent Programming*, 413–443. Springer.

Kogan, Alex, and Erez Petrank. 2012. "A Methodology for Creating Fast Wait-Free Data Structures." In *ACM SIGPLAN Notices*, 47:141–150. 8. ACM.

Marlow, Simon. 2012. "Parallel and Concurrent Programming in Haskell." In *Central European Functional Programming School*, 339–401. Springer.

Michael, Maged M, and Michael L Scott. 1996. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms." In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 267–275. ACM.

Morrison, Adam, and Yehuda Afek. 2013. "Fast Concurrent Queues for X86 Processors." In *ACM SIGPLAN Notices*, 48:103–112. 8. ACM.

"The Go Programming Language Specification." 2009. https://golang.org/ref/spec.

Yang, Chaoran, and John Mellor-Crummey. 2016. "A Wait-Free Queue as Fast as Fetch-and-Add." In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 16. ACM.