A Portable Lock-free Bounded Queue

Peter Pirkelbauer, Reed Milewicz, and Juan Felipe Gonzalez

University of Alabama at Birmingham
Birmingham, AL 35294
{pirkelbauer, rmmilewi}@uab.edu

Motorola Solutions
Birmingham, AL 35243
juanfelipe.gonzalez-gomez@motorolasolutions.com

Abstract. Attaining efficient and portable lock-free containers is challenging as almost any CPU family implements slightly different memory models and atomic read-modify-write operations. C++11 offers a memory model and operation abstractions that enable portable implementations of non-blocking algorithms. In this paper, we present a first scalable and portable lock-free bounded queue supporting multiple readers and multiple writers. Our design uses unique empty values to decouple writing an element from incrementing the tail during enqueue. Dequeue employs a helping scheme that delays helping in the regular case, thereby reducing contention on shared memory. We evaluate our implementation on architectures featuring weak and strong memory consistency models. Our comparison with known blocking and lock-free designs shows that the presented implementation scales well on architectures that implement a weak memory consistency model.

1 Introduction

FIFO queues are a fundamental data structure for many software systems. Due to their importance in multi-core computing, bounded and unbounded lock-free queues have been extensively studied [3,9,12,13,14,27,29,31]. Unbounded queues use dynamic memory management to store an arbitrary number of elements. Bounded queues are often implemented as circular buffers with a maximum storage capacity. Circular buffers do not require dynamic memory management and are well suited for embedded devices, real-time systems, operating systems, and environments demanding low space and performance overhead.

Developing portable nonblocking data-structures is difficult, because the available read-modify-write operations and implemented memory models differ substantially across architectures. For example, the x86 processor family features a fairly strict memory model that only allows the reorderings of loads before independent store operations [17]. A read-modify-write operation with infinite consensus number [9] on the x86 are the atomic compare-and-swap (CAS) instructions. CAS takes an address, an old value, and a new value. If the address contains the old value, the content is updated to the new value. CAS returns the

content stored at the address. Modern x86 CPUs offer single-word and double-word CAS instructions. ARM and PowerPC implement a weak memory model that allow reorderings of non-dependent reads and writes. A read-modify-write operation with infinite consensus number on ARM and PowerPC is the instruction pair Load-linked/Store-conditional (LL/SC). LL reads a value from a memory location. SC writes a value to the same location under the condition that no one has modified that location meanwhile. Many hardware implementations of LL/SC can spuriously fail under certain conditions.

The 2011 revision of the ISO C++ programming language, C++11 [11][30], specifies a concurrent memory model. The memory model defines the behavior for data-race-free-0 (DRF0) programs [2]. Critical sections can be synchronized using mutual exclusion locks. The standard lock's semantics guarantees that any update to shared memory inside the critical section is visible to subsequent threads acquiring the same lock. Portable lock-free programming is supported by atomic types. They offer a unified interface to a system's read-modify-write operations and fine-grain control over when memory updates become visible to other threads. This paper presents a lock-free circular queue based on C++11's atomics. The choice of C++11 trades better portability for some performance (i.e., when an architecture makes stronger guarantees than C++11). Our implementation relies on unbounded counters, which are available for all practical purposes on any 64bit and many 32bit systems (double word atomic operations). A single enqueue operation relies on two acquire/release operations, a single dequeue operation uses two sequentially consistent operations and two acquire/release operations. An evaluation on three different architecture families (x86, Power8, and ARMv8) demonstrates the portability and scalability of our queue.

The contributions of these paper are: (1) a portable lock-free bounded queue using the relaxed memory model; (2) performance analysis of available bounded queues on a variety of architectures; (3) a descriptor that uses a short common path; (4) an ABA-free solution that does not require free-store management.

The paper is organized as follows: §2 offers background on lock-free programming and describes the C++11 concurrent memory model based on known lock-based bounded queue implementations. §3 discusses our implementation. §4 evaluates our approach in terms of correctness and performance. §5 presents related work and §6 provides a summary and outlook on future work.

2 Background

Many multi-threaded systems rely on mutual exclusion locks (mutex) to protect critical sections and shared resources. Deadlock, livelock, priority inversion, and termination safety pose serious challenges to the design, implementation, and lifetime of such systems [9].

Lock-freedom, Linearizability, and History: Lock-free algorithms avoid those problems by not using locks. Instead they rely on a set of atomic read-modify-write operations such as CAS and LL/SC. Lock-free systems guarantee that one

out of many contending threads will make progress in a finite number of steps. The principle correctness condition of nonblocking systems in a sequentially consistent memory model is linearizability [10]. An operation of a concurrent object is linearizable if it appears to execute instantaneously in some moment of time between the time point of its invocation and the time point of its response. This definition implies that for any concurrent execution there must exist an equivalent sequential execution of the same operations. The ordering of operations in the sequential history has to be consistent with the real-time order of invocation and response in the concurrent execution history. For relaxed memory models, Batty et al. [1] propose the notion of a *history* as a semantic correctness condition. A history records operations, call and return events from multiple partial orderings, and their interactions as defined by the memory model. An abstract data structure is stated in terms of a history. A data structure implements the abstraction if it can produce the same history.

The *ABA problem* [18] is fundamental to many lock-free algorithms and occurs when a thread T reads value a from a memory location m. Other threads set m to b and then back to a. Thread T is unaware of the intermediate change and its CAS operation to replace a with a value v will spuriously succeed.

C++11 memory model: C++11 distinguishes between data operations and synchronization operations. Memory locations subject to data races have to be of atomic type. A data race is defined as two or more concurrent memory accesses to the same memory location, where at least one of them is a write [25]. Atomic operations include load, store, and read-modify-write operations.

Programmers can exercise fine-grain control over memory ordering by tagging atomic operations Table 1. The default semantics on atomic memory operations is sequential consistency, `memory_order_seq_cst`, which establishes a total order among all sequential consistent atomic operations. Compiler and hardware are not allowed to perform any intra-thread reordering on sequentially consistent operations. All non-atomic operations are partially ordered with respect to the sequentially consistent atomic operations on the same thread. Maintaining sequentially consistency is expensive on many modern architectures. The tags `memory_order_release` and `memory_order_acquire` form pairs that establish a *synchronizes-with* relationship between a thread A that stores a value v to an atomic memory location l and a thread B that loads that value v from l. Release/acquire guarantee that any operation in thread A that happens before storing v to l can be observed in B after it has loaded v from l. The memory model allows reorderings with subsequent operations on the storing thread and preceding operations on the loading thread as long as the intra-thread dependencies allow the reordering. The C++11 locks use release and acquire semantics for `lock` and `unlock` operations [2]. On the PowerPC acquire/release can be implemented by a light-weight barrier [24]. Release/consume (`memory_order_consume`) consistency is similar to release/acquire except that it restricts memory ordering to data dependent loads in a consuming thread. The consume tag does not require synchronization instructions on most architectures [17]. Relaxed consistency, `memory_order_relaxed`, does not guarantee any memory ordering and does

```
struct CircularBuffer {
  bool enq(int elem);
  pair<int, bool> deq();

  size_t tail;
  size_t head;
  int buf[N];
};
```

```
bool enq(int elem) {
  if (tail == head+N)
    return false;
  buf[tail%N] = elem;
  ++tail;
  return true;
}
```

```
pair<int, bool> deq() {
  if (tail == head)
    return make_pair(−1, false);
  int res = buf[head%N];
  ++head;
  return make_pair(res, true);
}
```

Fig. 1. Sequential bounded queue

not establish a synchronizes-with relationship. Memory operations can also be ordered by using atomic thread fences that can be tagged similarly.

Loads and stores tagged with relaxed or release/acquire may not become visible to all threads at the same time [33]. Consider two producers, A and B, that write to variables a and b respectively. A consumer Y could see the store to a but not b, while another consumer Z could see the store to b but not a. The ISO C++11 standard stipulates that an implementation must make an atomic store available to other threads in a finite amount of time and that all threads agree on a modification order of a memory location.

CAS and LL/SC are supported through `compare_exchange_strong` ($\mathtt{CAS}_S$) and `compare_exchange_weak` ($\mathtt{CAS}_W$). The two operations take a reference to an old value (`old`), a new value (`val`), and two memory ordering tags for success and failure respectively. If the atomic object equals the old-value, CAS sets it `val` and returns `true`. Otherwise the functions return false, and there semantics decay to a correspondingly tagged load operation that stores the value in `old`.

Descriptions of the C++11 memory model and more subtle details are described by the C++11 standard [11], Boehm and Adve [2], and Williams [33].

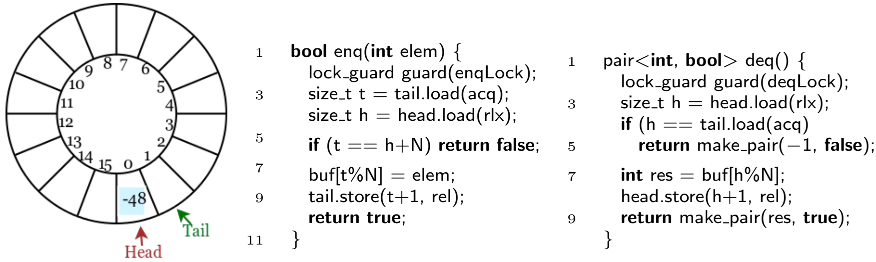
Circular Buffer and the C++11 memory model: The bounded queue in Fig. 1 stores integer values in an array `buf`. The maximum capacity of the data structure is given by the constant `N`. `enq` adds an element to the tail; `deq` reads an element from the head's position. Both operations `enq` and `deq` are nonblocking, as they return an error code when their preconditions not-full and not-empty are not met. The diagram in Fig. 2 sketches the data structure and shows a bounded queue of size 16 containing one element. The queue is empty when `head` equals `tail` and full when `head`$+N$ equals `tail`. This implementation uses unbounded counters for `head` and `tail`, which are practically available on any 64bit architecture or any 32bit architecture supporting double-wide CAS.

A concurrent implementation can be derived by adding a mutex per bounded queue object. The use of a single lock for all operations avoids data races on shared data members (i.e., `head`, `tail`, `buf`) by serializing accesses. Although C++11 mutexes uses acquire/release semantics, the use of a single lock makes the queue quasi sequentially consistent.

To attain a higher degree of concurrency, the queue can be modified to support a single concurrent enqueuer and a single concurrent dequeuer. The key idea is that `buf` and `tail` are updated only in `enq`, and `head` in `deq`. `enq` stores the element before it increments the tail, thereby making the buffer element available to a concurrent `deq` operation. Likewise `deq` copies the buffer element to a local

Memory Order	Relationship among/between operations
seq_cst	atomic operations are totally ordered non-atomic operations are partially ordered with respect to sequentially consistent atomic operations on the same thread.
release/acquire	form a pair on the stored/loaded value. This guarantees that the reading thread sees all memory updates in the storing thread that occurred before the store tagged with release.
release/consume	form a pair on the stored/loaded value. This guarantees that the reading thread sees all memory updates in the storing thread that occurred before the store and where there is a dependency relationship to the loaded value
relaxed	no synchronization relationship

Table 1. Memory Order Tags in C++11




```
1  bool enq(int elem) {
     lock_guard guard(enqLock);
3    size_t t = tail.load(acq);
     size_t h = head.load(rlx);
5    if (t == h+N) return false;
7    buf[t%N] = elem;
9    tail.store(t+1, rel);
     return true;
11 }
```

```
1  pair<int, bool> deq() {
     lock_guard guard(deqLock);
3    size_t h = head.load(rlx);
     if (h == tail.load(acq)
5      return make_pair(−1, false);
7    int res = buf[h%N];
     head.store(h+1, rel);
9    return make_pair(res, true);
   }
```

Fig. 2. Bounded Queue **Fig. 3.** Concurrent enqueue **Fig. 4.** Concurrent dequeue

variable, before it makes the empty memory location available to a concurrent `enq` by incrementing the `head`. Fig. 3 and Fig. 4 illustrate the additional changes for a two-lock implementation under the C++11 memory model. Both operations `enq` and `deq` use separate locks named `enqLock` and `deqLock` respectively. The locks order the execution of two enqueues and two dequeues, but there is no ordering between enqueue and dequeue. Consequently, `head` and `tail` have to be atomic. When `enq` updates `tail` in Line 9 to x, the use of release semantics makes all buffer stores up to location $x - 1$ observable to other threads that read x from `tail`. Note that `deq`'s store operation incrementing `head` uses release semantics (Line 9) though there is no load of `head` tagged acquire. Release is necessary to prevent reordering with loading the value from the buffer (Line 8). `buf` does not need to be atomic, because there exists no data race on buffer elements.

Since `enq` and `deq` use two distinct locks, there is no more implicit order relationship between these operations. Hence, accessing `head` and `tail` may load "stale" values, where a dequeue (enqueue) could perceive the buffer to be spuriously empty (full).

3 Design and Implementation

The major challenges for the design of a lock-free bounded queue are: (1) an enqueue operation has to update the buffer location and the `tail` seemingly atomically. (2) Since the bounded queue's storage is reused, delayed threads are prone to the ABA problem. Software solutions exist in the form of multi compare and swap (MCAS) [5]. MCAS relies on a bit to distinguish a regular value from a descriptor identifier. In a first phase, MCAS replaces all affected mem-

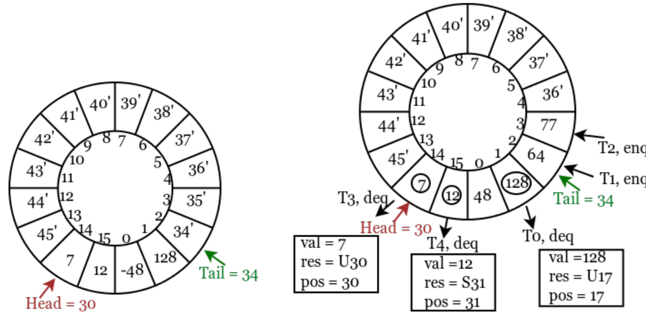


Fig. 5. Lockfree Queue **Fig. 6.** Concurrent execution

ory locations with a descriptor identifier specifying old-values and new-values for all M memory locations. If this succeeds, MCAS exchanges the descriptor identifiers with the actual values in a second phase. If one memory location was updated before, the first phase fails, and the second phase restores the original values. Any interrupting thread that reads a descriptor identifier will help the original thread finish the MCAS (phase one and two) before it carries out its own operation. Helping threads execute the same sequence of operations along a common path. Thus, helping diminishes parallelism and increases contention on the same memory locations.

3.1 Design

This section describes the high-level design of a lock-free bounded queue for integers. The bounded queue can be adapted for some type T, as long as atomic operations on T are available and we can distinguish a value of T from special entries, such as empty values and descriptors. The implementation on integers reserves a bit for marking special entries.

We address the identified problems the following way: In a first step, we decouple the two memory updates of the `enq` operation. This is achieved through storing unique empty values in empty queue locations[22,23]. Each empty value is a representation of the next `tail` position that will produce a successful enqueue operation. An empty value is marked using two bits (one to distinguish it from data and one to distinguish it from a dequeue descriptor.) An enqueue operation uses $\texttt{CAS}_S$ to replace the expected empty value with the new value. This scheme prevents delayed threads from the ABA problem. A delayed enqueue can never succeed spuriously by overwriting a valid value or a later empty value because either the buffer location contains the expected empty value or other threads have enqueued (and possibly dequeued) at that location.

Fig. 5 shows the class definition and Fig. 6 shows a graphical view of the data structure. The queue contains four elements stored between `tail` and `head`. The other entries are empty (indicated by the '). Each empty value represents the next `tail` position where an enqueue will be successful. A thread that attempts to enqueue a new value, reads the `tail` and attempts to store the new value there (using $\texttt{CAS}_S$). A thread that succeeded in storing the new element at position

p will attempt to set the `tail` to the following position $p + 1$ as long as `tail` contains a position that is less than $p + 1$. If the $\texttt{CAS}_S$ to store a new value is unsuccessful, another thread must have succeeded at that location. In this case, the enqueuing thread will retry at the next position $p+1$ as long as that position is less than `head +` *buffersize*.

The use of empty values shifts the burden of updating two memory locations atomically to dequeue. `deq` needs to update `head` and store the empty value associated with the next successful enqueue operation at that location. To this end, our dequeue operation utilizes descriptor identifiers. A descriptor identifier is marked using two bits (one to distinguish it from data and one to distinguish it from an empty value). A dequeue operation proceeds along the following steps:

1. Set up a descriptor and use $\texttt{CAS}_S$ to store a descriptor identifier at the dequeue location. The descriptor contains information on the previous value, the position, and a result flag. The result flag stores three states (undecided, success, fail). If the $\texttt{CAS}_S$ is successful, the dequeue operation is in progress and the descriptor can only be replaced by either a new empty value (success), or the previous value (failure).
2. Validate the operation by checking that the current `head` is at the location stored in the descriptor. This step updates the descriptor's flag.
3. If successful, move the `head` forward.
4. If successful, store the next empty value; otherwise restore the previous value.

The order of these steps prevents the ABA problem. At any given `head` location, a dequeue can only succeed once. In any successful dequeue, the descriptor is stored in the buffer when $\texttt{head} < \texttt{loc} < \texttt{head} + N$ and it remains in the buffer until `head` has advanced beyond the descriptor position. Any validation of another dequeue at the same location will fail.

An optimized helping scheme: If a thread gets delayed, other threads will read the descriptor identifier and help complete the delayed thread's operation to guarantee progress. A straight-forward implementation of helping would lead to contention on the shared common path. We remedy this problem (a) by delaying helping until a thread has found a valid dequeue location and (b) by employing a helping scheme that minimizes the common path.

When a dequeuing thread t_0 reads another thread's descriptor whose location is the same as the thread's own dequeue position, then t_0 will attempt to dequeue from the next location (if elements are still available) before it helps the delayed thread. After a thread has successfully stored a descriptor it will validate all descriptors between the current `head` and the new `head` position. The validation step will *only* set the flag of each active descriptor. After these descriptors have been validated, the thread updates `head` and replaces its descriptor with the next empty value. This technique increases the chance that other threads finish their dequeue operation before they need help and reduces the contention on the buffer and `head`. Multiple dequeue operations can be in-flight concurrently.

A thread finding a descriptor that does not correspond to the expected `head` location helps the delayed thread finish all three steps (validation, `head` update,

```
// chkENTRY = 1, chkSTATE = 3
// stMAXBITS = 2, stVALID = 0
// stEMPTY = 1, stWIP = 3

// Circular Buffer
struct CircularBuffer {
  bool enq(int elem);
  pair<int, bool> deq();

  atomic<int> head;
  atomic<int> tail;
  atomic<int> buf[N];
  DeqDesc ti[THREADS];
};
// work in progress data
struct WipData { // local copy of descriptor
  int desc; // identifier
  int res; // result flag
  int pos; // position
  int val; // original entry
};

bool enq(int val) {
  int pos = tail.load(rlx);

  while (pos < head.load(acq) + N) {
    atomic<int>& entry = buf[pos%N];
    int elem = empty_val(pos);

    if (entry.CAS_S(elem, val, rel, rlx)) {
      update_counter<rlx>(tail, pos+1);
      return true;
    }

    atomic_thread_fence(cns);
    if (is_val(elem))
      pos = pos+1;
    else if (in_progress(elem)
             && !this_was_delayed(elem, pos))
      check_descr(elem);
    else
      pos = tail.load(rlx);
  }

  return false;
}
```

```
// dequeue descriptor
struct DeqDesc {
  atomic<int> res;
  atomic<int> pos;
  atomic<int> val;
};

pair<int,bool> deq() {
  int descriptors[NUM_THREADS];
  int threadid = this_thread_id();
  int pos = head.load(rlx);

  while (pos < tail.load(rlx)) {
    atomic<int>& entry = buf[pos%N];
    const int elem = entry.load(rlx);
    int pvel = elem;

    if (is_val(elem)) {
      const int descr = make_descr(pos, elem, threadid);
      const bool succ = entry.CAS_S(pvel, descr, rel, rlx);

      if ( succ
        && check_descr( descr, get_descriptor(threadid),
             WipData(descr, descr, pos, elem), descriptors)))
        return make_pair(elem, true);
    }

    if (in_progress(pvel)) {
      atomic_thread_fence(cns);
      if (eqpos_descr_counter(pvel, pos)) {
        descriptors[pos % THRDS] = pvel;
        pos = pos + 1;
      }
      else if (this_was_delayed(pvel, pos))
        pos = head.load(rlx);
      else
        check_descr(pvel);
    }
    else
      pos = head.load(rlx);
  }

  return make_pair(−1, false);
}
```

Fig. 7. Lockfree bounded queue: enqueue and dequeue

and descriptor replacement). This is needed in order to remove an invalid descriptor from the buffer and restore the existing value to be dequeued.

3.2 Implementation

In a lock-free bounded queue threads can concurrently attempt to read and write the `head`, `tail`, and the buffer elements (`buf`). All data members are modeled with atomic types. Each entry in the `buf` array is either a valid value (lowest bit is 0) or a special value (1). Special values can symbolize either an empty entry (lowest two bits 01) or a work descriptor (11) of an going `dequeue` operation.

Upon queue construction, each buffer element is initialized with a unique empty value. The empty values are a function of the `tail` position that will produce a successful enqueue operation at that location. Our implementation left-shifts the position by 2 and adds the tag for an empty value (Fig. 8 function

```
template <memory_order mo>
void update_counter(atomic<int>& ctr, int pos) {
  int curr = 0;
  while ((curr < pos) && !ctr.CAS_S(curr, pos, mo, mo));
}

int make_descr(int headpos, int entryval, int threadno) {
  DeqDesc& ti = get_descriptor(threadno);
  const int descr = encode_descr(headpos, threadno);

  ti.res.store(descr, rlx);
  ti.val.store(entryval, rlx);
  ti.pos.store(headpos, rlx);

  return descr;
}

void decide(DeqDesc& ti, WipData& wip, int descr) {
  if (ready(wip)) return;

  int min = head.load(seq);
  const int valid = (wip.pos >= min);
  const int result = (wip.desc & ~stWIP) | valid;
  const bool succ = ti.res.CAS_S(wip.res, result, rlx, rlx);

  if (succ) wip.res = result;
}

bool check_descr(int descr) {
  DeqDesc& ti = get_descriptor(descr);

  return check_descr(descr, ti, load_threadinfo(ti, descr));
}

bool check_descr(int descr, DeqDesc& ti,
                 WipData wip, int* descriptors = nullptr) {
  if (inconsistent(wip)) return false;

  decide(ti, wip, descr);
  if (in_progress(wip)) return false;

  return complete(descr, wip, descriptors);
}

int validate_descr(int descr) {
  return validate_descr(get_descriptor(descr), descr, true);
}

bool in_progress(int v) {
  return (v & chkSTATE) == stWIP;
}
```

```
void help_delayed(int pos, int* descriptors) {
  if (!descriptors) return;

  int p = pos - 1;
  int h = head.load(rlx);

  while (p >= h) {
    int entryval = descriptors[p % THRDS];

    validate_descr(entryval);
    p = p - 1;
    h = head.load(rlx);
  }
}

int validate_descr(DeqDesc& ti, int descr, bool valid) {
  const int result = (descr & ~stWIP) | valid;
  const bool succ = ti.res.CAS_S(descr, result, rlx, rlx);

  return result;
}

bool this_was_delayed(int descr, int thispos) {
  return thispos <= get_descriptor(descr).pos.load(rlx);
}

bool complete( int desc, const WipData& wip,
               int* others) {
  const bool succ = success(wip);

  if (succ) {
    help_delayed(wip.pos, others);
    update_counter<seq>(head, wip.pos+1);
  }

  const int entryval =
        (succ ? empty_val(wip.pos+1) : wip.val);
  atomic<int>& entry = buf[idx(wip.pos)];

  entry.CAS_S(desc, entryval, rlx, rlx);
  return succ;
}

bool inconsistent(const WipData& wip) {
  return ((wip.desc ^ wip.res) & ~stWIP) != 0
      || !eqpos_descr_counter(wip.res, wip.pos);
}

int empty_val(int pos) {
  return ((pos << stMAXBITS) | stEMPTY);
}
```

Fig. 8. Lock-free bounded queue: auxiliary functions

`empty_val`). Unique empty values decouple the enqueue's write of the buffer element from advancing the `tail`. The empty values also help decide, when a `enq` operation is delayed and needs to resynchronize with `tail`. In addition, unique empty values prevent the ABA problem of delayed enqueue operations.

The descriptor: `deq` employs a descriptor announcing the operations. Our implementation uses one reuseable descriptor for each thread and does not need lock-free memory reclamation. The descriptor consists of four entries `desc`, `res`, `pos`, and `val` store the original descriptor (used for validation), result of the `head` validation, the position, and the read value respectively.

Encoding of the descriptor's identifier: The task of the descriptor's identifier is to allow other threads find the descriptor. In addition, the identifier has to be sufficiently unique in order to guard against the ABA problem. Hence, we use an encoding similar to a technique described by Luchangco et al. [16]. In our implementation, the two lowest bits are reserved to encode the kind of entry (value, empty-value, or dequeue descriptor) and later the result of the started `deq`. The following n bits are reserved to encode the thread id. Our implementation uses

eight bits for that. The remaining bits store the lowest bits of the corresponding buffer position. Assuming 256 threads, the descriptor prevents ABA if a helping thread is not delayed for more than 2^{54} and 2^{22} dequeue operations on 64bit and 32bit systems respectively. On 64bit systems, the number of supported threads can be easily increased without sacrificing ABA safety.

Validating the descriptor: After the descriptor has been stored, thread will validate the descriptor. Descriptor validations query the current `head`. The dequeue is valid if the descriptor's position is equal or higher than `head`. Validation will update the descriptor's `res` field using a $\mathtt{CAS}_S$ with the result. The next step validates all other in-flight descriptors between $\mathtt{head} \leq \mathtt{pos}(\mathtt{desc}_{\mathtt{other}}) < \mathtt{pos}(\mathtt{desc})$. In order to prevent an ABA problem introduced through reuse of descriptors, we encode the result field with the descriptor's position. The two lowest bits are reserved to store the result (undecided, success, fail), while the remaining upper bits are tagged with the descriptors location. Any helping thread that gets delayed cannot erroneously update the `res` field, because any time the descriptor gets set up for a new location, the `res` tag changes. Consequently, any delayed thread that attempts to update that field will fail.

3.3 Detailed description of enqueue and dequeue

The enqueue operation (Fig. 7): `enq` uses two loads to determine a range $([\text{tail}, \text{head} + N))$ where new values could be stored. `head` is loaded using acquire (Line 26) which synchronizes with the sequentially consistent update of `head` in a dequeue operation. This guarantees that the oldest entry in a buffer that the $\mathtt{CAS}_S$ in Line 30 can see is a valid dequeue descriptor that was stored before `head` was advanced. The next step is to enqueue the new element by using $\mathtt{CAS}_S$ with an empty value (tail+N′) at the *first* available position. The $\mathtt{CAS}_S$ can be unsuccessful because the buffer may contain one of the following values:

- *a valid value*: in this case, the enqueue will be reattempted at the next buffer location. `acquire` on `head`'s load guarantees that the enqueue cannot load a "stale" valid value that was stored at that location earlier. Only if loading `head` is delayed by more than $2N$ operations, enqueue could load a valid value. However, in this case either the enqueue fails earlier because the condition $\mathtt{pos} < \mathtt{head} + N$ does not hold, or also `tail`'s value is stale, in which case we could not enqueue at that location anyway. Thus, enqueue does not skip a valid location due to a "stale" value and stores the new element at the first available position.
- *descriptor*: in order to use a descriptor, we need to access its values. To this end, the thread fence `consume` is inserted before handling descriptors (Line 35). If the enqueuing operation was delayed, we load the new `tail` and reattempt the enqueue. Otherwise, the enqueue helps finish and remove the descriptor (Line 40) and retries at the same location.
- *an empty value*: in this case the empty value must be more recent than the enqueues expected empty value. Hence, we reload `tail` (Line 42) and retry the enqueue.

If no available buffer location was found, enqueue terminates. Otherwise, the `tail` location is advanced to one past the enqueued position.

The dequeue operation (Fig. 7): `deq` declares a local circular buffer where it stores the descriptors of concurrent dequeue operations (Line 9). The local buffer avoids reading the buffer a second time during the helping phase of the descriptor validation. The `thread_id` of a thread is needed to encode the descriptor (Line 10). `deq` uses a local variable, `pos`, to iterate through the buffer elements until a valid element is found and a dequeue can be attempted. (`while` loop in Line 13). Lines 14-16 create an atomic reference of the buffer element and copy its current content into a mutable (`pval` - previous value) and non-mutable (`elem`) variable. The load is tagged with relaxed, because the content of the buffer will be confirmed through a successful $\texttt{CAS}_S$ operation later. If the loaded element is a dequeable value, `deq` sets up its own unique descriptor identifier and attempts to store it in the buffer (Lines 19,20). If the $\texttt{CAS}_S$ fails, some other thread has modified the element at `pos` before (or another store to that location became visible). In this case $\texttt{CAS}_S$ falls back to a relaxed load. It can return one of the following values in `pval`:

- *descriptor identifier*: The atomic consume fence (Line 29) guarantees the memory ordering between reading the descriptor identifier and the descriptor data. If the descriptor indicates another dequeue at the same location, then `deq` stores the other descriptor identifier in its local buffer and attempts to dequeue from location `pos+1`. If the dequeue locations are different, then one of the two threads got delayed. If `deq` is delayed, it rereads `head` (Line 35), otherwise it helps the other thread finish the operation and remove the descriptor from the buffer (Line 39).
- *regular or empty value*: some other thread must have made progress at the same location and `deq` will resynchronize with `head`.

If the $\texttt{CAS}_S$ succeeds, `deq` invokes `check_descr` to complete the pending operation. If `check_descr` succeeds, `deq` returns the dequeued value (Line 25). The successful $\texttt{CAS}_S$ (Line 20) uses release to guarantee store ordering between the descriptor data and the buffer update.

Using descriptors (Fig. 8 functions `make_descr`, `check_descr`, `decide`, `complete`): `make_descr` sets up a descriptor for the current dequeue operation and returns a unique identifier to the descriptor. The descriptor encodes the result field with the same unique identifier. Thus, the result field can be updated using $\texttt{CAS}_S$ to store whether the operation succeeded. This avoids the ABA problem when a helping thread gets delayed. The unique identifier encodes the current position and thread id. `make_descr` uses relaxed stores that will be *released* when the descriptor's identifier is stored to the buffer.

The `check_descr` functions' task is to validate the descriptor and complete the operation in progress. The unary version is called when a thread is helping another thread. In this case it decodes the descriptor's identifier to find a specific thread's descriptor (Line 30-32) and creates a local copy `wip` (work in progress). The main `check_descr` (Lines 37–42) makes sure that the local descriptor copy is

consistent, calls `decide` to validate the dequeue at that location, and if successful calls `complete` to finish the operation.

`inconsistent` (Lines 42-45) validates that the loaded data is consistent. To this end, the descriptor identifier is compared with the identifier loaded from the descriptor, and the loaded position is compared with the encoded position in the identifier. If the thread that started the dequeue has already finished the operation associated with the loaded identifier and started another `dequeue` operation, this consistency check will fail and helping is no longer necessary.

`decide` validates the operation. It stores the outcome of the dequeue operation into the descriptor's `res` field and the local copy (`wip`). Line 19 returns immediately, if the outcome of the operation has already been decided. Line 20 loads the current head location. The read is sequentially consistent and synchronizes with the `head` update in `complete`. Using sequentially consistent memory ordering is necessary for two reasons: (1) storing the descriptor's identifier in the buffer and updating `head` are two independent operations that need to be ordered – we need to guarantee that the descriptor identifier was stored in the data structure before `head` was updated; (2) reads and writes to `head` need to be globally ordered; otherwise the read may return some "stale" value, which could lead to an erroneous validation. Lines 22 tests that `head` is smaller than the dequeue location. If `head` is larger some other thread has already dequeued an element from this position. This happens when some thread interleaves its `deq` between the time when `deq` read the `head` or when the dequeue's `pos` got out of sync with `head`. Line 30 attempts to store the result in the descriptor. If this fails, some other thread had modified the result in the meantime. If unsuccessful, `wip.pos` contains the result, otherwise we set it (Line 26).

`complete` finishes a started dequeue. If the operation is valid, `complete` helps other threads validate their descriptor and then advances `head` (Lines 30–32). Updating `head` is a sequentially consistent operation that synchronizes with loading `head`'s value in `decide`. Lines 35-38 either restore the old value, or store the next empty value corresponding to `pos+N`. If this operation fails, some other thread must have executed the same compare and exchange. Note it is also possible that a helping thread was delayed between `check_descr`'s consistency check and the execution of `decide`'s compare and exchange and therefore loaded a result corresponding to a later operation. In this case, executing `complete` will not modify the state of the bounded queue. The counter update will be unsuccessful, because some other thread must have succeeded on the current position and set `head` to a value greater than `pos`. Similar the compare and exchange in Line 46 will fail, because the descriptor's identifier must have been removed from the buffer before.

Example: Fig. 6 shows the concurrent execution of five threads on the lock-free bounded queue. Two threads, T1 and T2, have successfully stored their values in the buffer. Since the `tail` is not yet updated, T2 must have attempted to store its value at location 35, but failed because T1 had succeeded before. T1 and T2 will attempt to set `tail` to 35 and 36 respectively. If T2 succeeds first, T1 will skip the `tail` update. Three threads (T0,T3,T4) are dequeuing. T0 was

```
1  pair<int, bool> deq() {
     int pos = head.load(relaxed);
3    while (pos < tail.load(acq)) {
       int res = buf[idx(pos)].val.load(relaxed);
5      if (head.CAS_S(pos, pos+1, rel, rlx))
         return make_pair(res, true);
7    }
     return make_pair(−1, false);
9  }
```

Fig. 9. Lock-free dequeue in a hybrid implementation

delayed between the time it read `head` and the time it placed the descriptor at location 17. Since `head` has been moved past 17, the validation against `head` will fail, and the original value (128) will be restored. T3 and T4 are dequeuing from a valid location. Since both threads are still active and `tail` has not been updated, T4 must have seen T3's descriptor at location 30 before placing its own descriptor at location 31. Then T4 will validate the descriptor against `head`. T4's `res` field is already set to success. Since T3's descriptor is undecided (`res` is U), T4 will help T3 validate its descriptor before updating `head`. After descriptor validation, T3 and T4 will attempt to update `head`, and then replace their descriptor identifier with the next unique empty values (46' and 47').

Local descriptor buffer: A dequeue uses a local circular queue to store descriptors that require validation. In our implementation, the local queues capacity equals the number of threads. Since a thread can have at most one valid descriptor in the queue, this size enables a `deq` to search for an element that can be dequeued without having to consider a full local queue. The local queue size could be reduced while retaining lock-free properties. In this case, a `deq` would need to help other threads when its local buffer becomes full, before it can continue searching for a valid queue element.

3.4 Alternative implementations

We also experimented with other bounded queue designs. One of them is a *hybrid* implementation. They key observation for the hybrid design is that dequeue needs to update only `head`. This allows for a simple lock-free implementation displayed in Fig. 9. Reading `tail` uses acquire semantics in order to establish a happens before relationship with the previous store to `tail`. This guarantees that the buffer updates up to entry `tail-1` are visible. Line 5, stores the buffer entry in a local variable. If the $\texttt{CAS}_S$ in Line 7 succeeds, this `deq` returns the dequeued element. Otherwise, dequeue is retried at the most recent `head` position. The implementation of enqueue continues using a lock and is the same as in Fig. 3. Note, since there can be a data race between delayed dequeuing threads and an enqueue operation, the buffer elements have to be of atomic type.

4 Evaluation

This section discusses correctness tests, model checking, and performance evaluation of the circular queue implementation.

System	Sockets	Cores / Socket	Threads / Core	Total HW Threads	Frequency (Ghz)	OS	Compiler
IBM Power 8	2	10	8	160	3.4	Linux 3.10	xlc 13.1.3
Intel E5-2660	2	10	1 (HT disabled)	20	2.6	Linux 3.18	icc 15.0.3
AMD Opt. 6128	1	8	1	8	2	Linux 3.2	gcc 4.9
Snapdragon 410	1	4	1	4	1.2	Android 5.1 (32 bit)	clang 3.8

Table 2. Description of test systems

Correctness Tests: We performed multiple correctness tests on systems exhibiting weak and strong memory models and on single and dual socket configurations. We tested various load scenarios and used different buffer sizes (starting from a size of 2). We tested both with the same value across all enqueues (potentially prone to ABA) and with unique values for all enqueue operations. After reaching a quiescent state, we compared the history of each thread with the state of the queue to verify that the number of successful enqueues and dequeues agrees with the initial buffer size and state of the queue.

Model Checking: To test the validity of our implementation, we used Norris and Demsky's stateless model checker, CDSChecker [21]. CDSChecker provides its own implementation for atomics and threading. The CDSChecker exhaustively searches all possible interleavings and memory operation results (in particular for the relaxed memory model). The CDSChecker records an execution scenario and prints an execution trace if a violation was detected. CDSChecker can report the presence or absence of data races, deadlocks, uninitialized atomic loads, and user-provided assertion failures. We modified our implementation slightly to make it compliant with the CDS framework. The most significant change was the replacement of `consume` with `acquire` tags, since CDS Checker does not yet support `consume` semantics. We devised scenarios where threads played the roles of enqueuers and/or dequeuers and attempted to modify the bounded queue concurrently. At the end of operations, we validated the state of the queue, and that the elements remaining in the data structure were consistent with the number of successful operations. An exhaustive examination of all combinations of two operations in two threads and some three thread cases revealed no bug.

Performance Evaluation: We used strong scaling to test the performance of each implementation. For each implementation, we varied the number of threads to complete a fixed 40 million operations and a buffer size of 1024 which was initialized with 512 elements. After spawning the threads waited until all were ready to operate on the queue. Each thread alternated enqueue and dequeue operations. Thus, in a sequentially consistent implementation any operation must succeed. In relaxed implementations a failed operation was not counted and immediately retried. We ran the results twelve times, removed one outlier on each end, and report the average of ten runs. Performance experiments were conducted on four systems described in Table 2. We tested seven different algorithms. Singlelock uses a single lock to synchronize access to the data structure. Since only a single lock is used, the result is a sequentially consistent implementation. Doublelock refers to the two-lock queue presented in Fig. 3 and Fig. 4 which is similar to

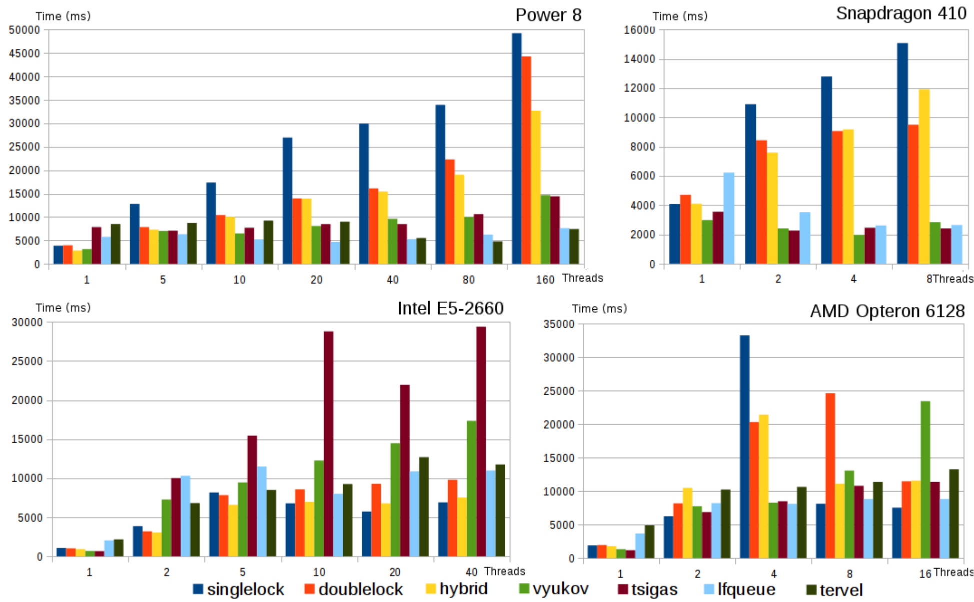


Fig. 10. Time of 40M successful operations on queues with a capacity of 1024 elements.

Linux SPSC implementation, where each role is protected by a lock; hybrid refers to the implementation outlined in Fig. 9 (note that these three lock-based implementations use a pthread mutex, which are implemented in terms of an OS Futex on Linux systems [4]. In most of our tests, futex locks scaled significantly better than standard spin-lock implementations.); Vyukov refers to the relaxed fine-grained lock implementation [32]; Zhang refers to Zhang and Tsigas' circular queue implementation (Note, to avoid the ABA problem and interference from a lock-free memory management technique, we enqueued unique elements); Tervel refers to the wait-free implementation by Feldman and Dechev [3,28]. Tervel was not tested on the Snapdragon running a 32bit OS. Lfqueue refers to the implementation discussed in this paper.

Fig. 10 shows the results on four test systems. Interestingly enough there is no single best approach. On architectures with a relaxed memory consistency model (ARM and Power), the lock-free and fine-grain locking implementations scale significantly better than the coarse-grain locking methods. On the Power system, Lfqueue outperforms other approaches from five threads ($\frac{1}{4}$ of cores) until 160 threads (number of hardware threads), with the exception of 40 threads, where Tervel is slightly faster. Beyond the number of available cores, Lfqueue and Tervel show significant better scalability than other approaches. On the ARM, the lock-free and fine-grain methods scale better than coarse-grain locking implementations. On x86 systems, the coarse-grain lock implementations scale slightly better than the Lfqueue, which in turn scales better than other lock-free and fine-grain locking approaches.

5 Related Work

The first non-blocking queue for a single enqueuer and single dequeuer was given by Lamport [14]. Other bounded queues also limit the number of concurrent enqueuers or dequeuers [7,8,15]. Stone [29] presents a lock-free bounded queue for multiple enqueuers and dequeuers using DCAS. DCAS atomically updates two independent memory location and is only available on older Motorola architectures. Shann et al. [27] present a lock-free bounded queue that relies on a double-word wide compare-and-swap (CAS2) instruction to prevent ABA problems. Tsigas and Zhang [31] present a scalable bounded queue that only uses single-wide compare-and-swap. Their implementation updates `head` and `tail` in steps of m, thereby reducing the contention on shared variables. To distinguish empty from full buffer entries, dequeue replaces a valid buffer element with a `null` value. In order to reduce the likelihood of ABA, their implementation reserves a bit of each buffer element, which is flipped after each enqueue/dequeue pair. This changes the ABA into an AB$\overline{\text{AB}}$A problem. To avoid ABA on stored elements, their buffer utilizes pointers to objects and dynamic memory management. Shafiei [26] presents a lock-free bounded queue that use collect objects to represent `head` and `tail`. The implementation compresses a 32bit value, index, old and new counters into a 64bit field. Shafiei's queue can store up to 2^{14} elements. A wait-free technique is given by Feldman and Dechev [3]. The wait-free construction requires dynamic memory management.

A portable implementation relying on the C++11 memory consistency model supporting multi-producer and multi-consumer is given by Vyukov [32]. His implementation uses per-element locks. When an element is locked a thread moves on to the next element. Frechilla [6] presents another multi-producer and multi-consumer queue based on two tail pointers. A thread gets a slot in the data structure by atomically incrementing the first tail pointer. After an element has been written, the thread will spin until it can increment the second tail pointer. Consumers dequeue up to the second tail pointer. These and similar designs are not termination safe and remain prone to priority inversion.

When the use of dynamic memory is feasible, unbounded queues can be implemented as lists or similar data structures. Multiple implementations exist [9,12,13,19,34]. Back-off and enqueue/dequeue matching techniques [20] for unbounded queues can reduce the contention on shared data.

6 Conclusion and Future Work

In this paper, we have presented a portable lock-free queue implementation for the C++11 memory model. Our approach enhances the descriptor based $MCAS$ design and allows for multiple concurrent descriptors without requiring lock-free memory management. We compared our implementation with other available lock-based and lock-free implementations on ARM, x86, and PowerPC architectures. Our approach scales particularly well on the Power8 architecture. On other systems, it scales as well as or better than other lock-free approaches. On

systems where coarse grain locking implementations scale well, we posit that our queue is a better choice when predictability and fault-tolerance become critically important.

As a next step, we plan to experiment with different back-off schemes to improve the performance under high-contention.

This work was partially funded by a Google Research Award, and by NSF grants CNS-0821497 and CNS-1229282. We thank the anonymous reviewers for their suggestions for improvements.

References

1. Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. *SIGPLAN Not.*, 48(1):235–248, jan 2013.
2. Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08*, pages 68–78. ACM, 2008.
3. Steven Feldman and Damian Dechev. A wait-free multi-producer multi-consumer ring buffer. *SIGAPP Appl. Comput. Rev.*, 15(3):59–71, October 2015.
4. Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast user level locking in linux. In *Linux Symposium in Ottawa*, pages 479–491, 2002.
5. Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), may 2007.
6. Faustino Frechilla. Yet another implementation of a lock-free circular array queue. http://www.codeproject.com/Articles/153898/Yet-another-implementation-of-a-lock-free-circular, Apr 2011. retrieved on March 3, 2013.
7. Ken Greenebaum and Ronen Barzel, editors. *Audio Anecdotes II: Tools, Tips, and Techniques for Digital Audio.* A K Peters/CRC Press, 2004.
8. Kjell Hedström. Lock-free single-producer - single consumer circular queue. http://www.codeproject.com/Articles/43510/Lock-Free-Single-Producer-Single-Consumer-Circular, Dec 2012. accessed on January 10, 2013.
9. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, revised 1st edition, 2012.
10. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
11. ISO/IEC 14882 International Standard. *Programming Language C++.* JTC1/SC22/WG21 - The C++ Standards Committee, 2011.
12. Christoph Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable k-FIFO queues. Technical Report TR2012-04, University of Salzburg, 2012.
13. Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *PPoPP'11*, pages 223–234, New York, NY, USA, 2011. ACM.
14. Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, apr 1983.
15. Patrick P. C. Lee, Tian Bu, and Girish Chandranmenon. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *ANCS '09*, pages 78–79, New York, NY, USA, 2009 2009. ACM.
16. Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '03, pages 314–323, New York, NY, USA, 2003. ACM.

17. Paul McKenney. Memory ordering in modern microprocessors (draft). http://www.rdrop.com/users/paulmck/scalability/paper/ ordering.2007.09.19a.pdf, sep 2007. retrieved February 20, 2013.
18. Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC '02*, pages 21–30, New York, USA, 2002. ACM.
19. Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *Euro-Par '03, LNCS volume 2790*, pages 651–660, 2003.
20. Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA'05*, pages 253–262, New York, NY, USA, 2005. ACM.
21. Brian Norris and Brian Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *OOPSLA '13*, pages 131–150, New York, NY, USA, 2013. ACM.
22. Peter Pirkelbauer. Non-blocking programming techniques, July 2013. University of Innsbruck, Invited Talk.
23. Peter Pirkelbauer. Portable non-blocking data structures, March 2013. University of Alabama, Invited Talk.
24. Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *PLDI*, PLDI '12, pages 311–322, New York, NY, USA, 2012. ACM.
25. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, nov 1997.
26. Niloufar Shafiei. Non-blocking array-based algorithms for stacks and queues. In *Proceedings of the 10th International Conference on Distributed Computing and Networking*, ICDCN '09, pages 55–66, Berlin, Heidelberg, 2009. Springer-Verlag.
27. Chien-Hua Shann, T. L. Huang, and Cheng Chen. A practical nonblocking queue algorithm using compare-and-swap. In 7th *International Conference on Parallel and Distributed Systems*, pages 470–475, 2000.
28. Damian Dechev Steven Feldman, Pierre LaBorde. Tervel, 2015. http://ucf-cs.github.io/Tervel/.
29. J. M. Stone. A nonblocking compare-and-swap algorithm for a shared circular queue. In *Parallel and Distributed Computing in Engineering Systems*, pages 147–152. Elsevier Science B.V., 1992.
30. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4 edition edition, 2013.
31. Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *SPAA'01*, pages 134–143, New York, NY, USA, 2001. ACM.
32. Dmitry Vyukov. Bounded MPMC queue. http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue, 2013. retrieved on May 21, 2016.
33. Anthony Williams. *C++ concurrency in action: practical multithreading*. Manning Publ., Shelter Island, NY, 2012.
34. Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *PPoPP '16*, pages 16:1–16:13, New York, NY, USA, 2016. ACM.