A Portable Lock-free Bounded Queue

Peter Pirkelbauer, Reed Milewicz, and Juan Felipe Gonzalez

No Institute Given

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 a range of architectures featuring weak and strong memory consistency models. Our comparisons with known blocking designs and another novel alternative lock-free design demonstrate that the presented implementation performs 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 [14][26],[25],[28],[13],[9],[12]. 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. The first concurrent bounded queue for a single reader and a single writer was described by Lamport [14].

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 [18]. 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 32-bit, 64-bit, and 128-bit wide 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, and the SC instruction fails despite no interleaving store to the memory location occurred.

The 2011 revision of the ISO C++ programming language, C++11 [11][27], specifies a concurrent memory model. The memory model defines the behavior for data-race-free-0 (DRF0) programs [4]. In order to synchronize access of shared resources, locks can protect critical regions. 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. In this paper, we will present a lock-free circular buffer based on C++11's atomics. Our implementation relies on unbounded counters, which are available for all practical purposes on any 64bit and many 32bit architectures (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, PowerPC, and ARMv7) demonstrates the portability and scalability of our circular queue implementation.

The contributions of these paper are: (1) a portable lock-free bounded queue using the relaxed memory model; (2) performance analysis of multiple bounded queue implementations on multiple platforms; (3) the lock-free queue uses unique values to represent empty entries; (4) an ABA-free solution that does not require free-store management.

The paper is organized as follows: §2 provides background information on lock-free programming and describes the C++11 concurrent memory model based on known lock-based bounded queue implementations. §3 discusses our implementation and §4.4 reasons about its correctness. The performance analysis is presented in §5. §6 discusses related work and §7 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.

2.1 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. This progress guarantee makes lock-free programs resilient against unexpected thread termination and priority inversion. 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. [2] propose the notion of a *history* as a semantic correctness condition. A history records operations and their interactions as defined by the memory model. Call and return events form multiple partial ordering. An abstract data structure is stated in terms of history between two operations. An specific data structure implements the abstraction if it can produce the same history.

The *ABA problem* [19] is fundamental to many lock-free algorithms and occurs when a thread T reads value a from a memory location m. Later 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 another value v will succeed despite the fact that the value was modified in the meantime.

2.2 The C++11 memory model

In this section, we will discuss the use of the C++ memory model based on available bounded queue implementations [9].

The 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 [23]. Programmers can exercise fine-grain control over memory ordering by tagging atomic operations. By default atomic operations use sequential consistency (tag `memory_order_seq_cst` to establish a total order among them. The tags `memory_order_release` / `memory_order_acquire` form a pair on the stored/loaded value. They guarantees that the reading thread sees all memory updates in the storing thread that occurred before the store tagged with release. The `memory_order_release` / `memory_order_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 to the loaded value. The tag `memory_order_relaxed` does not establish an ordering relationship. The read-modify-write operations CAS and LL/SC are supported through the atomic operations `compare_exchange_strong` ($\texttt{CAS}_S$) and `compare_exchange_weak` ($\texttt{CAS}_W$). The two `compare_exchange` operations take a reference to an old value (`old`), a new value (`val`), and two memory ordering tags (`succ` and `fail`) for the success and failure respectively. If the atomic object equals the old-value, `compare_exchange` updates

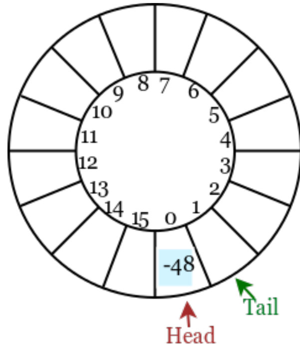

Fig. 1. Graphical overview

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

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

Fig. 2. Class definition

Fig. 3. Sequential bounded queue

it to the new value and returns `true`. Otherwise the functions store the current value in `old` and return `false`. Like LL/SC, `compare_exchange_weak` can spuriously fail.

Note, as a consequence of the total order on sequentially consistent operations, a store becomes visible to all concurrent threads at the same time. Likewise, read-modify-write instructions operate on a single, most recent, value of an atomic memory location. Conversely, loads and stores tagged with relaxed or release/acquire may not become visible to all threads at the same time [30]. 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 guarantees that all threads see the same modification order of each memory location. In addition, it stipulates that an implementation must make an atomic store available to other threads in a finite amount of time. For example, this disallows hoisting of atomic loads outside of loop bodies.

Descriptions of the C++11 memory model and more subtle details are described by the C++11 standard [11], Boehm and Adve [4], and Williams [30]. Table 1 gives an overview of the available tags and sketches their semantics.

2.3 Circular Buffer and the C++11 memory model

The bounded queue in Fig. 3 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. 1 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

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

Fig. 4. Enqueue

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

Fig. 5. Dequeue

Fig. 6. Sequential bounded queue

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	do not establish a relationship other than that a load reads a previously stored value

Table 1. Memory Order Tags in C++11

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. Any enqueue and dequeue operation would acquire the lock before executing the operation. The use of a single lock for all operations avoids data races on shared data members (i.e., `head`, `tail`, `buf`) by serializing accesses. The implementation of a standard mutex' `lock` and `unlock` operation guarantee that any update to the shared state can be consistently observed by any other thread that acquires the same mutex later. The use of a mutex prevents any true concurrency as at any given time there can only be a single thread that executes either `enq` or `deq`. Any concurrent thread will wait until the mutex becomes available.

To attain a higher degree of concurrency, `enq` and `deq` could use two distinct locks. Using two distinct C++11 locks, however, does not establish a proper ordering of data accesses in `enq` and `deq` operations.

The default semantics on atomic memory operations is sequential consistency (the tag's name is `memory_order_seq_cst`). This establishes a total order among all atomic operations that are tagged sequentially consistent. Compiler and hardware are not allowed to perform any intra-thread reordering on se-

```
bool enq(int elem) {
  lock_guard guard(enqLock);
  size_t t = tail.load(acquire);
  size_t h =
    loadRecent<relaxed>(head);

  if (t == h+N) return false;

  buf[t%N] = elem;
  tail.store(t+1, release);
  return true;
}
```

```
pair<int, bool> deq() {
  lock_guard guard(deqLock);
  size_t h = head.load(relaxed);
  if (h == loadRecent<acquire>(tail))
    return make_pair(−1, false);

  int res = buf[h%N];
  head.store(h+1, release);
  return make_pair(res, true);
}
```

Fig. 7. Bounded queue allowing concurrent reader and writer

quentially 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 particularly expensive on weakly consistent architectures, such as Alpha, ARM, and PowerPC. Release/acquire establishes 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 (the tags' names are `memory_order_release` and `memory_order_acquire`) guarantees 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. Release/acquire establishes a transitive relationship. Any updates obtained through a operation tagged require will be made available with a operation tagged release. The C++11 locks use release and acquire semantics for `lock` and `unlock` operations [4]. Thus the single lock based implementation presented in Fig. 3 guarantees that any updates by earlier threads to `buf`, `head`, and `tail` can be observed by a thread acquiring the lock later. On the PowerPC acquire/release can be implemented by a light-weight barrier [22]. Release/consume consistency is similar to release/acquire except that it restricts the observability of operations that preceded the store in thread A to memory locations that depend on the loaded value v in thread B. The consume tag does not require synchronization instructions on most architectures [18]. The relaxed consistency model does not guarantee any memory ordering and does not establish a synchronizes-with relationship. In addition to tagging atomic operations, memory operations can be ordered by using memory fences that can be tagged similarly.

The bounded queue implementation in Fig. 3 can be modified to support a single concurrent enqueuer and a single concurrent dequeuer.

As the code in Fig. 3 updates `buf` and `tail` only in `enq` and `head` in `deq`, the implementation allows for a single concurrent reader and writer. The key idea is that `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 variable, before it makes the empty memory loca-

tion available to a concurrent `enq` by incrementing the `head`. Fig. 7 illustrates the additional changes for a two-lock implementation under the C++11 memory model. Note that we abbreviate the memory ordering tags by omitting the common prefix `memory_order`. Both operations `enq` and `deq` use separate locks named `enqLock` and `deqLock` respectively. This change converts the former data operations on `head` and `tail` to synchronization operations. `enq` stores a value to `tail` in Line 10, while `deq` reads `tail` in Line 4. Consequently, the type of `head` and `tail` has to be atomic. Since `enq` and `deq` use two distinct locks, there is no more implicit memory order relationship. The acquire and release semantics of `lock` and `unlock` no longer guarantees that a value stored to a buffer location is in sync with the value loaded from that location, even though the update of the `tail` could be observed. Hence, stores and loads of `tail` use release/acquire semantics. `loadRecent`

To load the most recent value of the atomic variable that is not written within the same critical section (e.g., `enq`'s load of `head` in Line 5), both operations use a faux read-modify-write (RMW) operation encapsulated in a function `loadRecent`. The modify-part of the RMW operation will fail, but the read part will return the most recent value.

The transitive nature of release/acquire guarantees that an update of `tail` to x makes all buffer stores up to location $x-1$ observable to other threads that read x from `tail`. `deq`'s loads of `head` uses relaxed memory ordering, because the acquisition of `deqLock` guarantees that the last store to `head` is visible. `deq`'s store operation incrementing `head` uses release semantics to prevent reordering with loading the value from the corresponding buffer location. Note that `buf` does not need to be atomic, as there exists no data race on a specific buffer element. `enqLock` prevents two concurrently executing enqueues and synchronizes two consecutive enqueues. No data race involving two stores exists. `deq` cannot read an element at location x before a corresponding `enq` has updated the `tail` to point to location $x+1$. The acquire/release semantics on `tail` guarantees that the update to `buf[tail]` and the increment of `tail` are observed in order. This prevents data races involving a load and a store.

Other read-modify-write operations such as `fetch_add` also exist. Whether the manipulation of a specific type is lock-free can be tested with the member function (`bool is_lock_free()`).

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 object. In a first phase, MCAS replaces all affected memory locations with a descriptor object specifying old-values and new-values for all M memory locations. If this succeeds, MCAS exchanges the descriptor

```
// work in progress data
struct WipData {
  int desc; // descriptor's identifier loaded from buffer
  int res; // descriptor's identifier loaded from descriptor
  int pos; // position loaded from descriptor
  int val; // original entry loaded from descriptor
};

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

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.compare_exchange_strong(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;
}
```

```
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.compare_exchange_strong(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. 8. Lockfree bounded queue: enqueue and dequeue

objects 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 object 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 another type T, as long as atomic operations supporting T's size 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

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

template <memory_order mo>
void update_counter(atomic<int>& ctr, int pos) {
  int oldval = ctr.load(rlx);

  while ((oldval < pos) && !ctr.compare_exchange_strong(oldval, pos, rlx, rlx));

  atomic_thread_fence(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.compare_exchange_strong(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);
}

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

```
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.compare_exchange_strong(descr, result, rlx, rlx);

  return result;
}

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

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

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

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

  if (succ) {
    help_delayed(wip.pos, descriptors);
    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.compare_exchange_strong(descr, entryval, rlx, rlx);
  return succ;
}

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

Fig. 9. Lock-free bounded queue: auxiliary functions

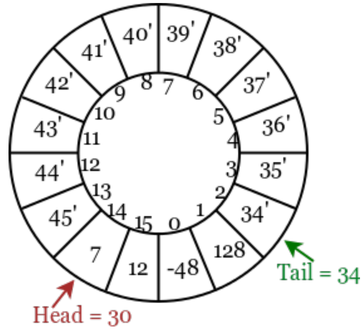

Fig. 10. Graphical Overview

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

  atomic<int> head;
  atomic<int> tail;
  atomic<int> buf[N];
  DeqDesc ti[THREADS];
};
```

Fig. 11. Class definition

Fig. 12. Lock-free bounded queue I

storing special values in empty queue locations. 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 and stored the next empty value) at that location in the meantime.

Fig. 12 shows the class definition and 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 in storing a new value at that location. In this case, the enqueuing thread will retry at the next position $p + 1$ as longs as that position is less than `head +` *buffersize*. Note, the update of the `tail` pointer to position $p+1$ may be delayed or may not execute at all if another thread succeeded in storing an element at a later position and already updated `tail`.

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 objects. A descriptor 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:

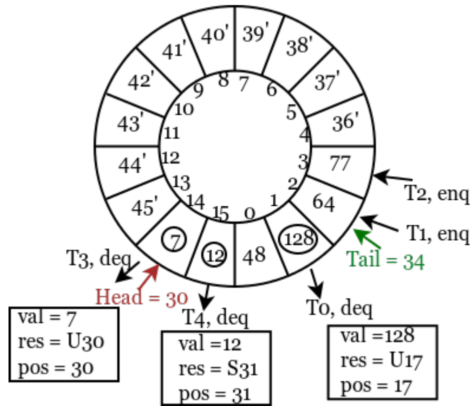

Fig. 13. Concurrent execution

Fig. 14. Lock-free bounded queue II

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 head $<$ loc $<$ head $+ N$ and it remains in the buffer until the `head` has been incremented 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 object 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). After a thread has successfully stored a dequeue descriptor it will validate it and then help other delayed threads validate their dequeue descriptor. A thread needs to validate all active descriptors between the current `head` and the new `head` position before

it can update `head`. The validation step will *only* set the flag of each active descriptor accordingly. After all descriptors in that range have been validated, the thread can continue with its own operation (`head` update and descriptor replacement). By delaying the validation step until a thread's own descriptor has been validated, we increase the chance that other threads finish their dequeue operation before they need help. This scheme reduces the contention on the buffer and `head` and allows multiple dequeue operations 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, 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) pointing to an on 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. **??** function `EmptyVal`). 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 dequeue operation (Fig. 8 function `deq`): Similar to `enq`, `deq` uses a local variable `pos` to identify the current location, where it attempts to read a value (Line 2). In Line 6, the current buffer element is loaded into `entryval`. If `entryval` is a valid value, `deq` attempts to replace this element with the next empty value and increment the `head`. This two stores need to be executed seemingly atomically.

The descriptor: Similar to MCAS [5], `deq` employs a descriptor object announcing the operations. After the descriptor is placed in the data structure, the dequeue position used to place the descriptor is validated against the most recent `head` position. The dequeue is correct if the position is ahead of the `head`. Lastly the descriptor is replaced either by the next empty value (success), or by the original value (failure).

Our implementation does not allocate the descriptor objects dynamically and does not require memory reclamation techniques. Instead, our approach reserves one descriptor object for each thread. The descriptors will be reused every time a thread dequeues an element. 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. [17]. 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 queries 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 $\texttt{CAS}_S$ with the result. The next step validates all other in-flight descriptors between $\leq \texttt{pos}(\texttt{desc}_{\texttt{other}}) < \texttt{pos}(\texttt{desc})$. In order to prevent an ABA problem introduced through reuse of descriptor objects, 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 the enqueue and dequeue

The enqueue operation (Fig. 8): `enq` uses a local variable, `pos`, to iterate through the buffer elements until a store is successful or no more empty location is available (`while` loop in Line 19). In Lines 22-26, `enq` attempts to compare and exchange the expected empty value at the buffer location with the new element that is going to be stored. If the operation is successful, `enq` will make `tail` point to `pos+1` (Line 24). The $\texttt{CAS}_S$ (Line 22) uses memory order release if successful. This guarantees that a read of that location tagged with acquire can load elements depending on that value. The counter update is tagged relaxed, because no information is released. Interleaving threads may cause the $\texttt{CAS}_S$ to fail. In this case, the new buffer element at `pos` is returned in `entryval`. `enq` handles three different scenarios. First, `entryval` contains a value (Lines 29,30), thereby indicating that another thread has succeeded with an enqueue operation. `enq` will retry to enqueue at the next position. If that enqueue operation is successful, the `tail` can be updated to `pos+1` because the store to `pos` has already completed. Second, if `entryval` points to a dequeue descriptor (either a valid or invalid dequeue), `enq` will test if it was delayed by comparing the descriptor's position. If `enq` is not delayed, it help finish the dequeue operation and remove the descriptor from the buffer (Line 31–34). Afterwards `enq` retries the operation at the same location. Third, `entryval` could indicate an empty value that is out-of-sync with

FiXme Fatal: can the ops be reordered?

pos. In this case, enq will resynchronize pos with tail (Line 35 and 36). The thread fence tagged consume in Line 28 avoids reorderings between reading the descriptor (Line 22) and accessing the descriptor values (Line 31).

The dequeue operation (Fig. 8): deq declares a local buffer where it stores the descriptors of concurrent dequeue operations (Line 2). 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 3). 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 6). Lines 7–9 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 compare and exchange operation; the correct memory ordering with an element's content (in case of a pointer or descriptor) will be guaranteed through an accordingly tagged fence. 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 12,13). If that operation fails, some other thread has modified the element at pos before (or a store to that location became visible) and deq proceeds as if elem had not been a valid element. If the compare and exchange succeeds, deq invokes check_descr to complete the pending operation. If check_descr succeeds, deq returns the dequeued value (Line 18). A dequeue failure can have several reasons. (a) the new old value of pvel (obtained by the $\mathtt{CAS}_S$ in Line 13) is a descriptor object. If the two threads attempted to dequeue at the same location, then deq stores the other thread's descriptor 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, otherwise it helps the other thread finish the operation and remove the descriptor from the buffer. If pvel is a regular or empty value, then some other thread must have made progress at the same location and deq will resynchronize with head. If successful, the $\mathtt{CAS}_S$ (Line 13) uses release to guarantee store ordering between the descriptor field and the buffer update. If the $\mathtt{CAS}_S$ fails, the implementation falls back to a relaxed read. The atomic consume fence (Line 29) guarantees the memory ordering between the descriptor identifier and the descriptor data (accessed by check_descr). *Using descriptor objects (Fig. 9 functions make_descr, check_descr, decide, complete):* make_descr sets up a descriptor object for the current dequeue operation and returns a unique identifier to the descriptor. The descriptor encodes the result field with the unique identifier. Updating the result field uses $\mathtt{CAS}_S$ to replace the identifier with a flag that indicates 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 spe-

cific thread's descriptor object (Line 37–39) and creates a local copy `wip` (work in progress). The main `check_descr` (Lines 42–49) 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 .

FiXme Note: check call to in_progress(wip)

`inconsistent` (Lines 51–54) validates that the loaded data is consistent. To this end, the descriptor identifier is compared with the identifier loaded from the descriptor object, 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 and encodes the outcome of the dequeue operation into the descriptor object's `res` field also updates the local copy (`wip`). Line 26 returns immediately, if the outcome of the operation has already been decided. Line 28 loads the current head location. The read is a sequentially consistent read (`seq_cst`). This is necessary for two reasons (1) setting up a descriptor object and 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" old value, which could lead to an erroneous validation. Lines 29,30 test 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 31 attempts to store the result in the descriptor object. If this fails, some other thread had modified the result in the meantime. If successful, also `wip.pos` is updated .

FiXme Note: check why wip.pos is not updated if the thread checks itself

`complete` finishes a started operation by either restoring the buffer's old value in case `deq` was not started at a valid `head` position, or updating `head` and storing an empty value in the buffer. If the operation succeeded, `complete` helps other concurrent threads validate their descriptor objects and then advances the `head` (Lines 38–41). Updating `head` is a sequentially consistent operation that pairs with loading `head`'s value in `decide`. Lines 42–46 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.

Fig. 13 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

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

Fig. 15. Lock-free dequeue in a hybrid implementation

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 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').

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. 15. 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 $\texttt{CAS}_S$ that updates `head` by 1 in Line 7 is successful, 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 the two lock version in Fig. 7. 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.

By using unique empty values, we can devise another hybrid implementation that supports lock-free enqueues and uses locks to dequeue (or a lockfree single dequeuer). Finally, we implemented another lock-free bounded queue that uses helping for enqueues. Here, we employ a descriptor object similar to the described approach in §3.1 to achieve the atomic update of buffer element and `tail`. We noticed, that the absence of unique empty values complicates the im-

FiXme Note: cite scenario where this is useful; OS kernel?

plementation, because less information is available to determine when a thread gets delayed. Consequently, this implementation is outperformed by our main approach.

4 Correctness

We used several tests to validate the circular buffer on various architectures, we used the CDS Model Checker to verify the execution, and we will argue informally that our implementation is correct.

4.1 Correctness Tests

To test the correctness of data-structures we performed multiple tests on architectures exhibiting weak and strong memory models. This includes PowerPC a and B (Blue Gene), multi-socket and single socket x86 architectures, and multicore ARM systems. 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 we reached 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 number of elements in the queue. One test used alternating enqueues and dequeues with a buffer size that was equal to the number of threads. Since enqueues and dequeues alternate all operations in any linearizable implementation must succeed. On the G5 PowerPC architecture, the hybrid implementation experienced a small quantum of spurious unsuccessful enqueue and dequeue operations for relaxed loads from `head` and `tail`. Roughly 1 out of 1 million operations did not succeed.

4.2 Model Checking

To test the validity of our implementation, we used Norris and Demsky's stateless model checker, CDSChecker [**?**]. 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 data structure, 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 feasible buggy executions.

4.3 Correctness Considerations

Here we argue that our implementations are free of errors and fulfill lock-free correctness properties in the context of the relaxed memory model.

Progress: A concurrent object is lock-free if one out of many contending threads makes progress in a finite number of steps [9]. The critical points for progress are when `enq` and `deq` store a new value or a descriptor identifier respectively in the data structure. At each `tail` and `head` position, the first thread that executes the $\texttt{CAS}_S$ operation makes progress.

Termination Safety: The implementation is termination safe as long as there exist at least one dequeuer and enqueuer. Consider a dequeuing thread that terminates after it placed a (valid or invalid) descriptor identifier in the buffer. In case that the descriptor is valid, another dequeue will validate the descriptor, move the head past the descriptor (thereby enabling enqueues at that location), but leave the descriptor in the buffer. An enqueuing thread will find the descriptor and remove it from the buffer. Similarly, if a thread attempting an invalid dequeue terminates before the descriptor can be removed, any other operation will help validate the descriptor and then remove it from the buffer. If an enqueing thread terminates before it can increment tail, another enqueueing thread will see the new element and enqueue afterwards and move the tail, thereby making the first element visible. However, if an enqueue (or a valid dequeue) terminate, it requires another enqueue (or dequeue) operation to make the former operation visible. If no other enqueue (or dequeue) is running, the buffer will become full (or empty). In essence, this is consistent with the semantics of a sequential bounded queue, where the buffer becomes full (or empty) when complementary operations do not exist.

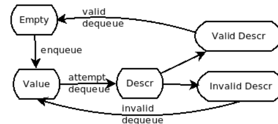

Fig. 16. Buffer element state transitions

Memory tag correctness: In this section, we argue that the memory tags are sufficient to keep the data structure consistent in the presence of non-sequentially consistent memory operations. Before discussing correctness, consider the state transitions of buffer elements in Fig. 16.

FiXme Note: mention "guarantees" upfront

enqueue: An enqueue uses two operations to determine a range ($[\text{tail}, \text{head}+N)$) where new values could be stored. `head` is loaded using acquire which synchronizes with the sequentially consistent update of `head` in a dequeue operation. This guarantees that the enqueue will see at least all valid dequeue descriptors that were stored in that range before `head` was advanced. The next step is to enqueue the new element by using CAS_S with an empty value (tail+N$'$) at the

first available position. The 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 $\texttt{pos} < \texttt{head} + N$ does not hold, or also `tail`'s value is stale, in which case we could not enqueue at that location anyways. Thus, enqueue does not skip a valid buffer location due to a "stale" valid 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. If the enqueuing operation was delayed, we load the new `tail` and reattempt the enqueue. Otherwise, the enqueue helps finish and remove the descriptor 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` 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.

dequeue: A dequeue operation is divided into three parts. First, dequeue finds a potential dequeue location and installs a descriptor object. To this end, it uses two relaxed loads to determine a dequeueable range ($[\text{head}, \text{tail})$). Then dequeue walks through that range and uses relaxed loads to identify the first dequeuable position. If a valid element is found, we use CAS_S to install a descriptor. One property of read-modify-write operations is that for consistency they modify most recent values (i.e., they cannot modify a stale entry). Thus, if the CAS_S was based on a stale value, the CAS_S fails and becomes a relaxed load operation and depending on the value, the same location may be retried. If dequeue finds another descriptor in the buffer, we distinguish three cases. (a) the descriptor is for the same dequeue location. In this case, we store the descriptor in an internal buffer and then dequeue continues with the next buffer location; (b) this thread is delayed, then we just load head and reexecute the dequeue; (c) the other thread is delayed, in which case, the dequeue helps remove the descriptor from the buffer before the dequeue is retried at the same buffer location. If the loaded value is an empty value, either the loaded range is stale, the loaded buffer value is stale, or the dequeuing thread got delayed. In any case the dequeue gets reexecuted.

Second, once the descriptor is installed we use one sequentially consistent load of `head` to validate the dequeue. The dequeue is valid if the dequeue location is greater than the current `head` value. If that is not the case, the dequeue may still have succeeded. In this case, another thread must have validated our descriptor before updating head. Thus, loading the descriptor value will see any

modifications that occurred before the `head` update. After validating our own descriptor, the dequeueing thread will validate all descriptors in the range of $[\text{head}_{\text{seq}}, \text{pos})$ and then advance `head` past the dequeued element. Since descriptor validation must not read a stale value, we need to use sequentially consistent tags for both the load of `head` and advancing `head`, which makes the two operations globally ordered.

Third, once the descriptor is validated, dequeue removes the descriptor and replaces it with the next empty value (or in case it failed with the previous entry).

4.4 Linearizability and History

Introduction: The traditional standard for correctness of concurrent algorithms is linearizability, but this property loses currency in the realm of weak memory models. Linearizable algorithms assume sequential consistency, but achieving this in practice requires expensive operations such as memory fences. As our library is implemented over the relaxed C++11 memory model and does not utilize sequentially consistent operations, the moniker of linearizable is not appropriate. Algorithms implemented in languages with weak memory models tend to yield constellations of partial orders on sets of operations; there may be many chains, that is, totally ordered subsets, but these chains may be disjoint. However, we can claim that our algorithm obeys a property that approximates the notion of linearizability.

In this section, we borrow the techniques of library abstraction for relaxed memory models as laid out by Batty et al. [2], which provides us with powerful formalisms for understanding C++11 concurrency. To take advantage of these formalisms, we must first provide a compatible, abstract specification of our algorithm, which we have done in Fig. 17. Although the specification differs from the actual implementation in certain key aspects, it contains all the behaviors of the original implementation. This falls in line with the Abstraction Theorem, which states that if a specification abstracts an implementation, then any behaviors of a client using the implementation are contained in the behaviors of a client using the specification. We assume we have a client which calls our abstract library (the specification) and whose threads execute arbitrary sequences of enqueue and dequeue operations with arbitrary inputs; this client is capable of producing any and all conceivable executions.

Notation: The semantics of a program are defined by the set of all valid executions of that program. An execution is defined as a tuple $X = (A, sb, ib, rf, sc, mo, sw, hb)$. A is the set of actions performed by threads, the remaining elements are various relations on A as defined by the memory model [2]. An individual action is defined as a tuple $a = (e, g, t, \varphi)$, where e is a unique identifier for the action, g is an atomic section identifier, t is the thread identifier, and φ is the effect of the action (e.g. *store* or *load*). Of these actions, we denote the *call* and *return* actions of the operations *enq* and *deq* as interface actions; these are the actions that concern the client code. We also distinguish specific actions where *enq* and *deq* operations are said to take effect as effect points.

From the perspective of the client, the interaction with the library amounts to a sequence of call and return actions. This view of the execution of the program is expressed as a history, produced by a history-extracting function `history`(X), where X is an execution of the program. A history is defined as a tuple $H = (A, G, D)$ where A a the set of call and return actions, and G and D are relations on A. G, which we refer to as the guarantees relation, is a restricted subset of the happens-before (hb) relation of the execution that captures all of the hb edges between call and return actions. D, meanwhile, is the denies relation, which contains the edges that the client cannot enforce because the semantics of the library render it impossible. As a consequence of the Abstraction Theorem, the abstract library must provide the same guarantees as the concrete library, but is allowed to deny less; this is because the abstract library may exhibit more behaviors than the concrete library. However, this only strengthens our proofs: if we can prove that the abstract library is safe, then we know that the concrete implementation must also be safe.

Proof `enq` *vs.* `enq`*:* Given an execution X and its history `history`$(X) = H = (A, G, D)$, let $e_0 = ((i, _, u, \texttt{call enq}(_)), (j, _, u, \texttt{return enq}(\texttt{true})))$ and $e_1 = ((k, _, v, \texttt{call enq}(_)), (l, _, v, \texttt{return enq}(\texttt{true})))$ such that $e_0, e_1 \in A$. We seek to prove that for any two such operations, if e_0 succeeded before e_1, then $e_0 \preceq_G e_1$.

First, we know that if e_0 and e_1 are performed by the same thread (i.e. $u \equiv v$), then it is trivial to show $e_0 \preceq_G e_1$, because the sequenced-before (sb) implies happens-before (hb). Therefore, we only need to consider the cases in which u and v are different threads.

When we examine the library-local semantics of the execution, we see that the effect point of an enqueue operation is when the thread successfully performs a double compare-and-swap to write its value and increment the head on line 31. This is analogous to the two-step process in the actual implementation, where a thread uses two compare-and-swap operations to replace the element (this is when the operation takes effect) and increment the tail (this may not succeed if another thread increments the tail before).

Since the `DCAS` updates the head and the memory contents at that position using release semantics, these updates form a chain under the modification-order (mo) relation. Whenever an enqueuing thread reads the value of the head as is done on line 15, even though it reads with weaker semantics, the axioms of the memory model require that the values it reads must respect the modification order. If the read occurs after a `DCAS`, then it must read the most recent value in the modification order. Otherwise, it is possible for the read to take its value from a previous write. However, the subsequent `DCAS` will synchronize with the release of the previous `DCAS` because the stale read will cause the `DCAS` to fail. Therefore, if e_0 succeeds in moving the head before e_1, either e_1 synchronizes with e_0 or with another, intervening enqueue operation which synchronizes with e_0; since the synchronizes-with relationship is transitive, e_1 synchronizes with e_0. The relation synchronizes-with implies happens-before. This means that $e_0 \preceq_G e_1$.

`deq` *vs.* `deq`*:* Let X and H be defined as in the previous subsection, and let $d_0 = ((i, _, u, \texttt{calldeq}(_)), (j, _, u, \texttt{returndeq}(_)))$ and $e_1 = ((k, _, v, \texttt{calldeq}(_)), (l, _, v, \texttt{returndeq}(_)))$ such that $d_0, d_1 \in A$. We seek to prove that for any two such operations, if d_0 succeeded before d_1, then $d_0 \preceq_G d_1$. As in the previous section, dequeue operations performed by same thread guarantee one another.

The effect point of dequeue operations can be found on line 57 when the thread successfully moves the position of the tail. The compare-and-swap operation and the preceding read are all executed with relaxed semantics, which is sufficient for updating a single counter. In that only one dequeuing thread can make progress at one time, and must take the value of the previous successor in order to progress, we are guaranteed that d_0 happens-before d_1 (or d_0 happens-before an intervening dequeue that, in turn, happens-before d_1) and therefore $d_0 \preceq_G d_1$.

`enq` *vs.* `deq`*:* Again, let X and H be as has been defined, and let $e = ((i, _, u, \texttt{call enq}(_)), (j, _, u, \texttt{return enq}(\texttt{true})))$ and let $d = ((k, _, v, \texttt{call deq}(_)), (l, _, v, \texttt{return deq}(_)))$ such that $e, d \in A$. We seek to show for any two such operations, either $e \preceq_G d$ or $d \preceq_G e$. Once again, two operations performed by the same thread guarantee one another.

Enqueue and dequeue operations coordinate with one another when they get the most recent values of tail and head on lines 19-25 and 44-50. This is done so that threads cannot enqueue into a full buffer or dequeue from an empty buffer. Since enqueues and dequeues are guaranteed to get the most recent values of tail and head, a successful e happens-before a later d and vice versa. Therefore, either $e \preceq_G d$ or $d \preceq_G e$.

Conclusion: We have established that for any execution X of the abstract specification, all operations captured by the history $H = \texttt{history}(X)$ are totally ordered by the guarantee relation. This means that all interface actions, the actions which concern the client, are totally ordered with respect to the happens-before relation. In the realm of relaxed memory concurrency, this is the closest approximation of sequential consistency that we can achieve.

5 Evaluation

Performance considerations:
Performance Tests: We used strong scaling to test the performance of each implementation. For each implementation, we varied the number of threads from two to sixty-four against a fixed 40 million operations and a buffer size of 128. Additionally, we varied the behavior of the threads from only enqueueing or dequeueing to alternating between enqueueing and dequeueing operations. We ran each experiment five times and took the average of the results. Performance experiments were conducted on three different architectures: x86 (AMD Opteron 8-core, Intel 20-core – 2 sockets with ten cores each and hyperthreading disabled), PowerPC (Power8 – 2 sockets with ten cores each and each core supporting up to eight hardware threads), and ARM (ARM Snapdragon 410 quad-core). In each case, the experiments were performed when no other users were using the

system. We tested four 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. Double lock refers to the two-lock implementation presented in Fig. 7 and is similar to Linux SPSC implementation, where each role is protected by a lock; hybrid refers to implementation outlined in Fig. 15; lockfree refers to the main implementation discussed in this paper. Zhang refers to Zhang and Tsigas circular queue implementation. We found that the lock-based implementations are particularly informative as they give us a benchmark to compare against the hybrid approach, which bridges the gap between lock-based and lock-free.

The most dramatic and revealing results for the non-blocking implementation are seen on the 24-core PowerPC. Here the described lock-free implementation best demonstrates its scalability, leveraging many threads across many cores to achieve the best results. Likewise, on the 12-core Intel architecture, we see that the lock-free implementation performs very well, although the hybrid implementation just barely outpaces it as the number of threads increases.

Trends similar to those seen on the 24-core PowerPC can be seen on the G5 PowerPC, Freescale QorIQ PowerPC, Intel dualcore, Intel Haswell, and ARM dualcore architectures, although, with fewer cores to take advantage of, the performance gap between lock-based and lock-free implementations is considerably smaller. However, as the number of threads increases, we observe that the ability of the lock-free implementations to manage contention gives them a slight edge over their competitors.

Finally, on the AMD architecture (graph not shown), we observed lock-based approaches performing better. Interestingly enough, the single-lock implementation outperforms the double lock implementation, which in turn is faster than the lock-free version. The less contention-prone hybrid version performs best. For performance, the environment favors simple, lock-based solutions, as threads can spin locally on locks; the descriptors, meanwhile, present a moving and ephemeral target, which strains performance due to latencies, high contention, frequent cache invalidation, and cache coherence traffic.

We also ran tests where we used threads that either enqueue or dequeue. We found that the obtained numbers are harder to interpret, because any truly concurrent implementation (double-lock, hybrid, and lock-free) shows performance imbalances where either `enq` or `deq` runs faster. Hence, the buffer either fills up, or empties out, leading to a high number of unsuccessful operations. Due to the buffer being circular, those unsuccessful operations increase contention on the slower buffer end. We were able to hand-tune the implementations by utilizing a back-off scheme, but such modifications are highly architecture dependent and cause bias towards specific implementations.

Comparison of Lock-free Implementation to Vyukov and Tsigas/Zhang Queue Implementations: To compare the performance of our lock-free implementation to a contemporary alternative, we also collected data on the performance of the Vyukov multi-producer, multi-consumer bounded queue implementation and the Tsigas and Zhang implementation, both of which are described in §6. The

Vyukov implementation is fine-tuned to maximize performance. It uses fine-grained locking and does not provide lock-free guarantees. In principal, the Vuykov queue is similar to our implementation, except that it does not provide helping, which makes it susceptible to threads delaying other threads and subsequent failures of operations. Testing Vuykov's queue with alternating enqueues and dequeues in four threads on an Intel Haswell (four cores) shows that not all enqueues and dequeues succeed. For example, with a buffer size of four, 3 out of 1000 operations fail. The number declines to 5 out 10000 and 1 out of 10000 when we increase the buffer size to 8 and 16 respectively. In terms of pure performance, we observe that, in general, the Vyukov implementation outperforms our lock-free implementation, although on Intel architectures, we note that the two are closely matched.

Meanwhile, our implementation of the lock-free queue of Zhang and Tsigas is competitive on the Haswell and ARM architectures, but hemorrhages cycles on machines with a higher number of cores. The reasoning for this is that the underlying algorithm, as it is given, makes no assumptions about the memory model of the implementation language or underlying architecture. Our C++11 implementation takes the conservative route and assumes that all atomic operations ought to execute with sequentially consistent semantics. On the other side, Zhang and Tsigas relies on memory management, such as hazard pointers [19], to avoid the ABA problem. In our tests, we stored unique integer values in the queue, and thus no memory management overhead was incurred.

Workload Distribution Tests: A balanced workload of 50% enqueue operations and 50% dequeue operations is ideal for performance because it maximizes the likelihood that the buffer will be neither empty nor full. In real world situations, however, this is rarely the case. Studying how the data structure behaves under different workloads gives two-fold benefit. First, it provides insight into how the data structure will behave "in the wild." Second, it lets us see how each operation (in this case enqueues and dequeues) performs under increased contention.

In each test, 16 threads performed random operations on a ringbuffer of fixed size, initially half full, over the course of 2000 milliseconds. We varied the distribution of enqueue and dequeue operations, from 25% enqueues and 75% enqueues to 70% enqueues and 30% dequeues. For a given implementation and workload distribution, the test was performed five times, and from these we calculated the average number of successful operations per millisecond. These results were normalized to the results of the ideal 50% enqueue / 50% dequeue distribution for a given implementation. Finally, we calculated the standard deviation for each implementation's set of normalized throughputs, which indicates extent to which each implementation is affected by changes in workload.

In Fig. 22, we see the throughput curves for each implementation on the Freescale 24-core PowerPC. It is worth noting that, for any implementation, its throughput curve is highly similar across all architectures. We can say that these graphs reveal information that characterizes the behavior of each implementation.

Table 2. Standard Deviations of workload distribution tests

	Intel	AMD	G5	24core PPC	ARM
Singlelock	.1427	.1394	.1552	.1637	.1583
Doublelock	.1681	.1782	.1682	.1755	.1550
Hybrid	.1563	.2486	.1636	.1812	.1537
Lockfree	.1002	.0865	.1209	.0495	.1356

The throughput curves for the single lock and double lock implementations are highly symmetric and sharper than the others. The symmetry is due to the fact that the lock-based implementations try to treat enqueuing threads and dequeuing threads equally, which means that workloads that are enqueue-dominated are treated the same as those that are dequeue-dominated. Meanwhile, the sharpness is caused by the time cost of threads waiting to acquire their lock only to fail in their operation due to the buffer being empty or full as a consequence of the imbalance in the workload.

Meanwhile, the hybrid implementation has an asymmetric throughput curve that is a blend of the lock-free and lock-based implementations. The hybrid with lock-free dequeue gently slopes downward as the percentage of enqueue operations increases. Ceteris paribus, if a client expects a specific range of unbalanced workloads, they will see more consistent performance by using a hybrid implementation tailored to their application than if they were to use a purely lock-based one.

Lastly, we see that the lock-free implementation is the flattest of all, an observation which we confirm in Table 2, which records the standard deviations of the measurement sets. This means that the lock-free implementation offers the most consistent performance across different workloads, due to its ability to mitigate both the cost of failure and the tension caused by contentious workloads.

6 Related Work

Numerous bounded queue implementations exist. The first wait-free 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], [15], [8]. Stone [26] presents a lock-free bounded queue for multiple enqueuers and dequeuers. His implementation relies on the availability of double compare-and-swap (DCAS). DCAS atomically updates two independent memory location and is only available on older Motorola architectures, but not on modern processors. Hardware transactions, such as on Intel's Haswell, will allow for such implementations. Shann et al. [25] present a lock-free bounded queue that relies on a double-word wide compare-and-swap (CAS2) instruction to prevent ABA problems. CAS2 only recently became more widely available on modern processor designs. Tsigas and Zhang [28] present a scalable bounded queue that only uses single-wide compare-and-swap (CAS). Their implementation updates `head` and `tail` in steps of m, thereby reducing the contention on shared variables. To dis-

tinguish 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 $\mathrm{AB}\overline{\mathrm{AB}}\mathrm{A}$ problem. To avoid ABA on stored elements, their buffer relies on dynamic memory management. Spurious CAS successes remain a problem when the buffer is almost full or empty and with relatively small queue lengths. Shafiei [24] presents a lock-free bounded queue that use collect objects to represent `head` and `tail`. In a collect object, each thread owns a field, where it has exclusive write access. The most recent value can be retrieved by scanning other threads' fields. The implementation compresses a 32bit value, index, old and new counters into a 64bit field, which allows for a buffer capacity of up to 2^{14} elements.

Some available concurrent libraries offer bounded queues [16][1], but they limit the number of concurrent readers, writers, or both. A portable implementation relying on the C++11 memory consistency model supporting multi-producer and multi-consumer is given by Vyukov [29]. His implementation uses per-element locks. When an element is locked a thread moves on to the next element. A thread holding a lock can delay other threads indefinitely. This makes the code simple (about 90 lines of code) and fast, but does not guarantee all lock-free properties (i.e., termination safety, resilience to priority inversion). 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. This design is not termination safe and 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 [20][9][13][12]. Back-off and enqueue/dequeue matching techniques [21] for unbounded queues can reduce the contention on shared data.

To reason about concurrent programs, Batty et al. [3] establish a mathematical model for the C++11 and C11 memory model. They axiomatized their concurrency model with Isabelle/HOL. Based on the Isabelle/HOL's code generation they derive a tool, CPPMEM, that generates and displays all possible executions and inter-thread relationships for short C++ programs.

7 Conclusion and Future Work

In this paper, we have presented a lock-free queue implementation and compared it with four lock-based implementations using the C++11 memory model. We described the testing of the lock-free implementation against potential defects and reasoned about the correctness of our implementation. We have tested the performance on ARM, x86, and PowerPC architectures. The presented lock-free approach outperforms simple lock-based approaches, because it allows for multiple concurrent operations, and another lock-free implementation due to

the use of the relaxed memory model. While a fine-grained locking approach is faster than our implementation, we posit that our queue is easier to design and test with because its behavior follows our intuition of linearizability. Hence, the lock-free implementation is suitable for systems where predictability and fault-tolerance is critically important.

As a next step, we plan to experiment with different back-off schemes to improve the performance under high-contention. Also, we plan to evaluate to what extend other available lock-free data structures can benefit from predictive empty values and the delayed helping scheme. The presented queues and tests can be obtained from `https://dl.dropboxusercontent.com/u/93927008/queue.zip`.

References

1. Samy Al Bahra. Concurrency kit. http://concurrencykit.org/, 2013. retrieved on February 21, 2013.
2. Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. *SIGPLAN Not.*, 48(1):235–248, jan 2013.
3. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM.
4. Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 2008. ACM.
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 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 *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, 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 *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, pages 78–79, New York, NY, USA, 2009 2009. ACM.
16. liblfds.org. Lock-free data structure library. http://www.liblfds.org/, 2013. retrieved on February 21, 2013.
17. 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.
18. 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.
19. Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30, New York, NY, USA, 2002. ACM Press.
20. Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *Euro-Par '03, LNCS volume 2790*, pages 651–660, 2003.
21. Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM.
22. Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 311–322, New York, NY, USA, 2012. ACM.
23. 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.
24. 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.
25. Chien-Hua Shann, T. L. Huang, and Cheng Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pages 470–475, 2000.
26. 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.
27. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4 edition edition, 2013.
28. 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.
29. Dmitry Vyukov. Bounded MPMC queue. http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue, 2013. retrieved on February 21, 2013.
30. Anthony Williams. *C++ concurrency in action: practical multithreading*. Manning Publ., Shelter Island, NY, 2012.

```
1   atomic int buf[];
    atomic int head;
3   atomic int tail;
    int bufSz;
5
    void init(int size) {
7     buf = new int[size];
      store_RLX(head,size);
9     store_RLX(tail,size);
      bufSz = size;
11  }

13  bool enq(int v) {
      atomic_sec {
15      int tailPos = load_RLX(tail);
        int lowerbound = tailPos − bufSz;
17      int headPos = load_RLX(head);

19      int max;
        if(lowerbound < headPos)
21        max = headPos;
        else {
23        CAS_RLX,RLX(head,headPos,headPos);
          max = headPos;
25      }

27      if (tailPos >= max) return false;

29      int oldValue = load_ACQ(buf[tailPos % N]);
        DCAS_REL,ACQ(buf[tailPos % n],oldValue,v,tail,tailPos,tailPos+1);
31    }
      return true;
33  }

35  int deq() {
      int output;
37
      atomic_sec {
39      int tailPos = load_RLX(tail);
        int headPos = load_RLX(head);
41      int lowerbound = headPos;

43      int max;
        if(lowerbound < tailPos)
45        max = tailPos;
        else {
47        CAS_ACQ,ACQ(tail,tailPos,tailPos+1);
          max = tailPos;
49      }

51      if (headPos >= max) return EMPTY;

53      output = load_ACQ(buf[tailPos % N]);

55      CAS_RLX,RLX(head,headPos,headPos+1);
      }
57
      return output;
59  }
```

Fig. 17. Abstract Specification

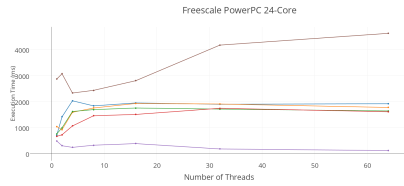

Fig. 18. Results on a Freescale 24-core PowerPC

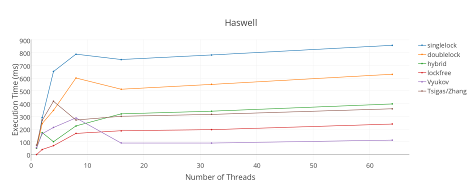

Fig. 19. Results on an Intel Haswell

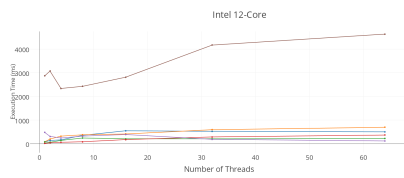

Fig. 20. Results an Intel 12-core

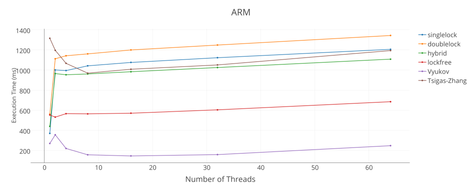

Fig. 21. Results on an ARM dualcore

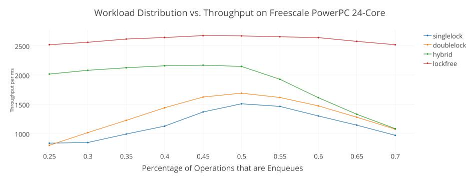

Fig. 22. Workload distribution test results on the Freescale PowerPC 24-core, with throughput left unnormalized to make each curve easily visible.