Apex.AI®

Lock-free Programming for Real-Time Systems

How Compare Exchange Will Become Your New Best Friend

Meeting C++ 2021

Matthias Killat

Lock-free Programming for Real-Time Systems

A journey of lock-free programming in C++

Real-Time Systems

System with multiple concurrent processes performing tasks

- tasks have priorities and deadlines
- tasks are often periodic
- Real-time OS (RTOS) guarantees priority-based scheduling of processes (and threads)
- RTOS cannot guarantee time bounds on completion of the user algorithms

Fair scheduling is necessary

- to meet the deadlines
- to allow system wide progress

Hard real-time

- not meeting deadlines is a system failure
- requires worst-case bounds on execution time of algorithms
- difficult to come by due to HW effects such as caching, branch prediction, interaction of threads etc.

Soft real-time

- not meeting deadlines is tolerable but degrades performance
- try to minimize number of exceeded deadlines or lateness
- degree of failure tolerance is application dependent

We focus on Soft Real-Time systems.

Concurrency in Real-Time Systems

Concurrency is used to increase scalability, performance and responsiveness

Requires synchronization of shared data

Synchronization primitives

- mutex
- semaphore
- condition variable
- promises and futures
- reader-writer locks

Problems

- blocking may lead to lockouts and deadlocks (due to programming error or partial system failure)
- priority inversion
  - high priority thread waits for resource held by low priority thread which was suspended
- starvation (due to program logic or scheduling)
- expensive context switches

Lock-Free Programming

Algorithms use no blocking primitives

Advantages
- no deadlocks, increased robustness against partial failures
- guarantee progress of at least one thread regardless of other threads
- no priority inversion
- generally fewer context switches (no blocking wait)

Disadvantages
- data synchronization becomes much harder
- more complex algorithms
- not necessarily faster than lock-based algorithms

Starvation of individual threads is still possible
- fair RTOS scheduling helps here
- wait-free algorithms avoid this (but are often infeasible)

Lock-Free Programming

OK, lock-free programming is supposed to help with some of the problems in real-time systems …

But how do we write lock-free algorithms?

Currently our friends for concurrent algorithms are mutexes, semaphores etc.

This is all well and good but does not help here ...

Time to make new friends then!

Lock-free Programming in C++

Atomics in C++

Atomics
- basic building block of lock-free algorithms
- available since C++11
- allows atomic access of data
    - only lock-free for small data (usually 64 bit)

Memory model
- synchronization of memory across threads
- imposes ordering with implicit memory barriers
- sequentially consistent (strongest) if unspecified
- out of scope in this talk

Read and Write operations
- load
- store

Read-Modify-Write operations
- fetch_add
- exchange
- compare_exchange

```
#include <atomic>

std::atomic<int> x{0};

x.store(73);

auto val = x.load();
// val = 73

auto old = x.fetch_add(1);
// old = 73

old = x.exchange(37)
// old = 74

++x;
// x = 38
```

Compare and Swap (CAS)

Available as compare_exchange_strong in C++

```
int expected;
bool result = x.compare_exchange_strong (
                        expected, desired);
```

- atomically swaps the current value with desired if current equals expected
- otherwise expected is set to the current value
- returns true if successful

- compare_exchange_weak can fail even if the condition is true
- may have superior performance on some HW

special HW instruction

- e.g. on x86: LOCK CMPXCHG

Major building block of most lock-free algorithms

CAS semantics

```
template <class T>
class atomic {
 T m_value;
  std::mutex m_mutex;

public:
  bool compare_exchange_strong(T &expected, const T
                                                   &newValue) {
    // enforce atomicity of the operation
    std::lock_guard<std::mutex> lock(m_mutex);

    if (m_value == expected) {
      m_value = newValue;
      return true;
    }
    expected = m_value;
    return false;
  }
};
```

CAS Loop

Fetch_multiply

Similar semantics as fetch_add

```
int fetch_multiply(std::atomic<int> &value, int multiplier) {
  int oldValue = value.load();

  do {
    // local computation of new value
    int newValue{oldValue*multiplier};
    if (value.compare_exchange_strong(oldValue, newValue)) {
      break;
    }
    // concurrent update occurred, retry until success
  } while(true);

  return oldValue;
}
```

Recurring pattern

- load current value
- compute new value locally
- try to update the old value atomically (CAS)
- retry if concurrent modification is detected

Starvation is possible but very unlikely in practice.

Only happens if other threads keep updating the value “faster” than fetch_multiply.

We can implement many lock-free operations with a CAS loop.

Exchange Buffer
A Case Study

Lock-free Data Exchange between Threads

Problem: Share data between threads without locks

- exchange data of a generic type T
- arbitrary number of concurrent readers and writers
- failure to write is allowed but should be rare and well-defined
    - e.g. out of memory

Solution: Lock-free Exchange Buffer

- buffer will just have one slot for data
- can be generalized to multiple slots to obtain e.g. a queue

Exchange Buffer

Exchange buffer
- may contain at most one element
- writing new data evicts any data in the buffer
- basically a queue/cache of size 1

Take operation
- destructive - data is removed from the buffer
- only one reader may succeed

Read operation
- non-destructive - data is still in the buffer
- multiple readers can read the data

Constraints
- lock-free
- avoid torn-reads or writes
- support types as general as possible

Interface

```
template <class T>
class ExchangeBuffer {
  // write value, replace existing value if any
  // return true if successful, false otherwise (out of memory)
  bool write(const T &value);

  // write value only if buffer is empty
  // return true if successful, false otherwise
  bool try_write(const T &value);

  // take value from buffer (buffer is empty afterwards)
  std::optional<T> take();

  // read value from buffer (value will still be in the buffer)
  std::optional<T> read();

};
```

Exchange Buffer

```
template <class T> class ExchangeBuffer {
private:
  // sufficiently  small for lock-free CAS or exchange operations
  std::atomic<T *> m_value{nullptr};

public:
  std::optional<T> write(const T &value);

  std::optional<T> take();
};
```

Store atomic pointer to the actual data

Assumptions

- T is copyable
- pointer size of 64 bit
- 64 bit CAS is available

We can only exchange data of limited size with CAS.

Write and Take

```
bool write(const T &value) {
  auto copy = new (std::nothrow) T(value);
  if (copy == nullptr) {
    return false; // no memory
  }
  auto oldValue = m_value.exchange(copy);
  if (oldValue) {
    delete oldValue;
  }
  return true;
}
```

```
std::optional<T> take() {
    auto value = m_value.exchange(nullptr);

    if (value) {
      auto ret =std::optional<T>(std::move(*value));
      delete value;
      return ret;
    }

    return std::nullopt;
  }
```

Exchange operations can be used to transfer memory ownership between threads.

Write and Take

```
bool write(const T &value) {
  auto copy = new (std::nothrow) T(value);
  if (copy == nullptr) {
    return false; // no memory
  }
  auto oldValue = m_value.exchange(copy);
  if (oldValue) {
    delete oldValue;
  }
  return true;
}
```

```
std::optional<T> take() {
    auto value = m_value.exchange(nullptr);

    if (value) {
      auto ret = std::optional<T>(std::move(*value));
      delete value;
      return ret;
    }

    return std::nullopt;
  }
```

not lock-free

- default memory allocator is not lock-free in general
- usually only thread-safe

We cannot use memory allocators that are not lock-free.

Interlude
Lock-free Memory Management

Lock-free Storage

```
template<typename T, uint32_t N, typename index_t = uint32_t>
class Storage {
public:

  void store_at(index_t index, const T &value);

  void free(index_t index);

  T *ptr(index_t index);

  T& operator[ ](index_t index);
};
```

Storage abstraction

- manages an internal array of memory to store objects of type T
- access the data by index
  - store (copy)
  - read
  - free

Basically an array with uninitialized memory for objects of type T

Lock-free Index Pool

```
template <uint32_t Size>
class IndexPool {
private:
  constexpr static uint8_t FREE = 0;
  constexpr static uint8_t USED = 1;
public:
  using index_t = uint32_t;

  IndexPool();

  // like malloc and free
  std::optional<index_t> get();

  void free(index_t index);

private:
  std::atomic<uint8_t> m_slots[Size];
 };
```

No we can build a special purpose allocator

- Storage contains the data
- stored data is indexed (index_t)
- access requires the index
- indices are managed by the IndexPool
  - similar to malloc and free
  - indices replace pointers
- thread which obtains the index has exclusive ownership of the storage slot and can safely read or write
- initially all slots are FREE

Indices are preferable since they usually require less space than pointers.

Prefer indices over pointers for CAS operations.

Lock-free Index Pool

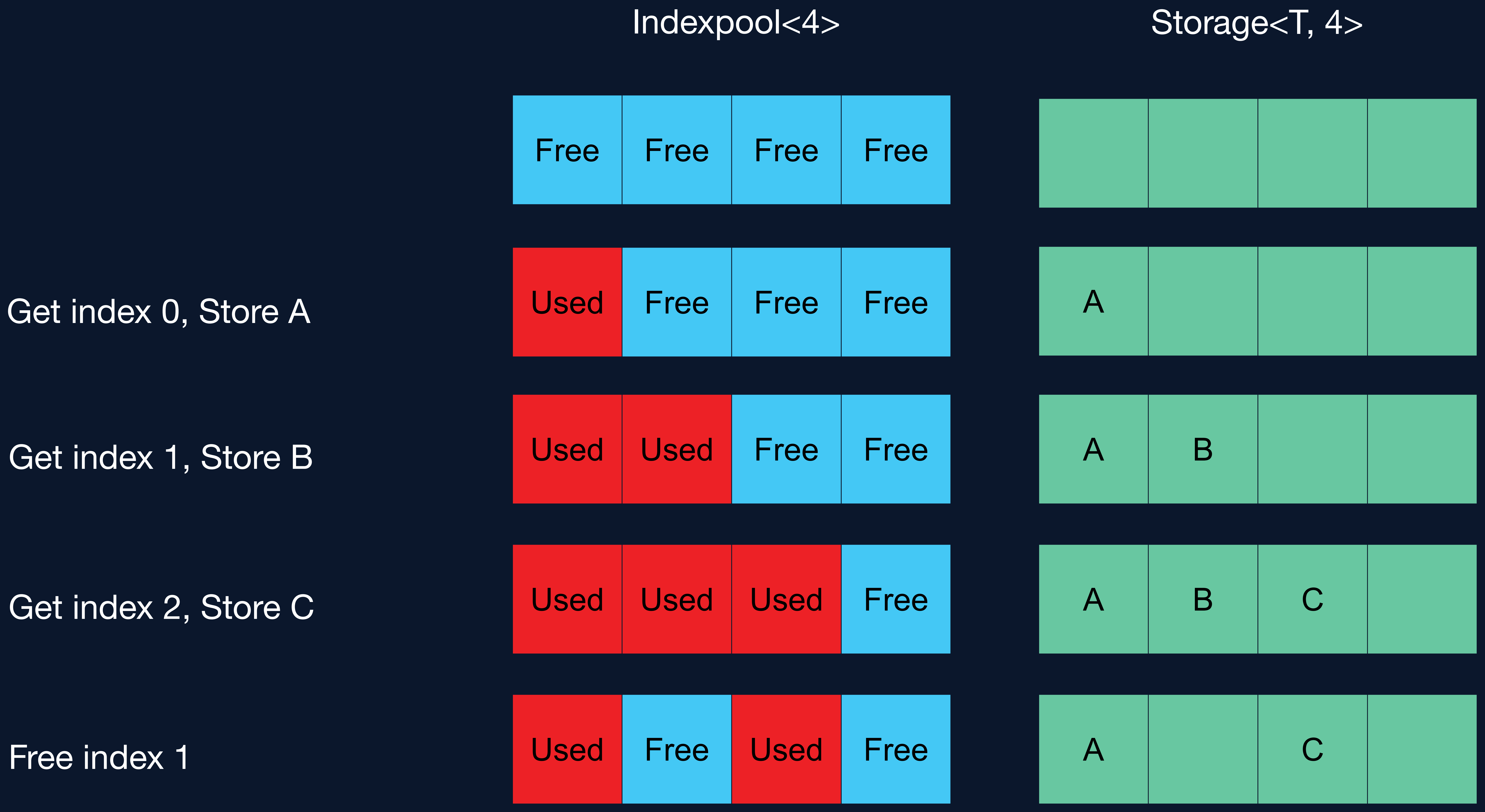
```
std::optional<index_t> get() {

    // single pass for simplicity
   for(index_t index = 0; index<Size; ++index) {
      auto expected = FREE;
      auto &slot = m_slots[index];
      if (slot.compare_exchange_strong(expected, USED)) {
         return index;
      }
   };
    return std::nullopt;
  }

void free(index_t index) {
    auto &slot = m_slots[index];
    slot.store(FREE);
 }
```

get
- try to obtain an index in a single pass
- fails if none is available

free
- return index we previously obtained
- if there is no misuse store suffices
  - no double free
  - do not free index we never obtained
- no one should write on a USED slot except the single thread which holds the index

Avoid CAS if store or exchange suffices.

Lock-free Memory Allocation



Writing is only allowed with ownership of the corresponding index.

ExchangeBuffer - Revisited

```
template <class T, uint32_t C = 4>
class ExchangeBuffer {
private:
  using storage_t = Storage<T, C>;
  using indexpool_t = IndexPool<C>;

   // indicates an empty buffer
  static constexpr index_t NO_DATA = C;
  std::atomic<index_t> m_index{NO_DATA};
  indexpool_t m_indices;
  storage_t m_storage;
public:
  bool write(const T &value);
  // ...
};
```

- atomic pointer replaced by an atomic index
- C controls the storage size for concurrent allocations
- C-1 concurrent writers are guaranteed to succeed
  - any more and some may fail (out of memory)
  - only holds if there are no concurrent readers
- memory is managed by Storage and IndexPool

Helper functions to manage memory

Free object corresponding to index

```
 void free(index_t index) {
      // call the destructor of T
      // return memory to storage
      m_storage.free(index);
      // return index to the pool
      m_indices.free(index);
  }
```

We need to manage the lifetime of objects we concurrently access.

ExchangeBuffer - Write

```
bool write(const T &value) {
  auto maybeIndex = m_indices.get();
  if (!maybeIndex) return false; // no index

  auto index = maybeIndex.value();
  m_storage.store_at(index, value);

  auto oldIndex = m_index.exchange(index);
  if (oldIndex != NO_DATA) {
    free(oldIndex);
  }
  return true;
}
```

- similar to version with pointers
- mechanism to get indices is lock-free
- complete algorithm is lock-free if copy constructor of T is lock-free

- we discard the old value
  - we could also return it
  - we would have to copy it from the internal buffer
  - essentially a write combined with a take

ExchangeBuffer - Take

```
std::optional<T> take() {
  auto index = m_index.exchange(NO_DATA);
  if (index == NO_DATA) {
    return std::nullopt;
  }

  auto ret = std::optional<T>(std::move(m_storage[index]));
  free(index);
  return ret;
}
```

- move or copy cannot be avoided since we reuse the internal buffer of m_storage

free(index)

1. calls the destructor at the corresponding slot
2. returns the index back to the pool

- the order is important
- index and memory can be reused by another write

ExchangeBuffer - Try_Write

```
bool try_write(const T &value) {
  auto maybeIndex = m_indices.get();
  if (!maybeIndex) return false; // no index

  auto index = maybeIndex.value();
  m_storage.store_at(index, value);

  index_t expected = NO_DATA;
  if(m_index.compare_exchange_strong(expected, index)) {
    return true;
  }
  // not empty, need to free the object we stored at index
  free(index);
  return false;
}
```

- only writes to empty buffer
- may fail due to
    - no index obtained (out of internal memory)
    - buffer full

ExchangeBuffer in Use

	Indexpool<4>				Storage<T, 4>				m_index
	Free	Free	Free	Free					No Data
Write U	Used	Free	Free	Free	U				0
Write V	Free	Used	Free	Free		V			1
Take returns V	Free	Free	Free	Free					No Data
Write X, Write Y	Used	Used	Free	Free	X	Y			1

Write X is not completed yet, i.e. CAS was not successful.

ExchangeBuffer - Take vs. Read

Taking data was rather simple with CAS or exchange ...

What about reading data without taking it out of the buffer?

This is considerably harder without locking primitives such as mutex.

Let’s try anyway and implement
std::optional<T> read();

ExchangeBuffer++
Reading Data in a Lock-free Way

ExchangeBuffer - Read

```
std::optional<T> read() {
    auto index = m_index.load();
    if (index == NO_DATA) {
      return std::nullopt;
    }

    return std::optional<T>(m_storage[index]);
}
```

ExchangeBuffer - Read

```
std::optional<T> read() {
  auto index = m_index.load();
  if (index == NO_DATA) {
   return std::nullopt;
  }
  // cannot read while it could change concurrently
  return std::optional<T>(m_storage[index]);
}
```

Data Race

Possible outcomes of the copy constructor

1. read the current value of m_storage[index]
2. torn read when reading a value that is currently changed
3. crash due to incomplete update (e.g. T= std::list)
4. crash due to deleted object
5. …

The object to be returned can be corrupted which leads to undefined behavior.

Be aware of data that can change concurrently.

ExchangeBuffer - Read

```
static_assert(std::is_trivially_copyable<T>::value);

std::optional<T> read() {
    auto index = m_index.load();
    if (index == NO_DATA) {
      return std::nullopt;
    }
    // cannot read while it could change concurrently
    return std::optional<T>(m_storage[index]);
}
```

Restrict the type to be trivially copyable

- we now effectively perform a memcpy
- data can still be concurrently modified
- may lead to torn reads
- but no crashes ...

From now on T is assumed to be trivially copyable.

Be aware of data that can change concurrently.

Read - 2nd Try

```
std::optional<T> read() {
   auto index = m_index.load();

   while (index != NO_DATA) {
    auto ret = std::optional<T>(m_storage[index]);


    if (m_index.compare_exchange_strong(index, index)) {
      return ret;
    }
    // if this failed the index changed concurrently
   }
   return std::nullopt;
  }
```

Retry read if we detect a concurrent change

- check whether index changed concurrently
- if so, the read is invalid and we retry

We still have a problem ...

ABA Problem

- index value may change back and forth while we are reading
- equal indices do not imply equal values

We can check whether the value is still the same with CAS.

ABA Problem

	Indexpool<4>				Storage<T, 4>				m_index
Read at index 1 is interrupted	Free	Used	Free	Free		U U			1 (A)
Write X X	Used	Free	Free	Free	X X				0 (B)
Write Y Y	Free	Used	Free	Free		Y Y			1 (A)
Read at index 1 completes	Free	Used	Free	Free		Y Y			1 (A)

Possible read results

U U	outdated, but OK	Y Y	OK	U Y	torn read - invalid

The index is the same when the read completes but the stored value has changed.

ExchangeBuffer - Read

Ok, this seems hard because we need to detect concurrent modifications while reading somehow ...

We could try counting whenever we write a value.

We need the ABA count ...

Extending Index with an ABA-Counter

```
template <class T, uint32_t C = 4>
class ExchangeBuffer {
 // … same definitions as before …
 struct tagged_index {
   tagged_index(index_t index) : index(index) {}
   tagged_index(index_t index, uint32_t counter)
       : index(index), counter(counter) {}
   index_t index;
   uint32_t counter{0};
  };
  // is the tagged index small enough to work with CAS?
 static_assert(std::atomic<tagged_index>::is_always_lock_free);
  // contains ABA counter as well as index
  std::atomic<tagged_index> m_index{NO_DATA};
 // … same interface as before ...
};
```

Instead of the index_t before we use a tagged index

- similar to tagged pointers
- contains ABA counter
- small enough for atomic lock-free operations
  - this is why we prefer indices over pointers
  - can optimize and use even more bits for the counter

ABA counter

- increment the counter in every write operation
- detect concurrent changes by comparing the counter with CAS
- works up to counter wrap-around
- more counter bits are better

We can add additional information to atomic data used in CAS.

Write with ABA Counter

```
bool write(const T &value) {
  auto maybeIndex = m_indices.get();
  if (!maybeIndex) return false; // no index
  tagged_index newIndex{ maybeIndex.value() };
  m_storage.store_at(newIndex.index, value);
```

Basically the same logic as before

- get an index if any is available
- combine it with a counter in a tagged_index
- the counter is uninitialized (will be set later)
- store the value at the corresponding slot

Write with ABA Counter

```
bool write(const T &value) {
  auto maybeIndex = m_indices.get();
  if (!maybeIndex) return false; // no index
  tagged_index newIndex{ maybeIndex.value() };
  m_storage.store_at(newIndex.index, value);
  tagged_index old = m_index.load();
  do {
    newIndex.counter = old.counter + 1; // increment ABA counter
    if (m_index.compare_exchange_strong(old, newIndex)) {
      if (old.index != NO_DATA) {
        free(old.index);
      }
      return true;
    }
  } while (true);
  return true;
}
```

We now need CAS to modify the index and counter atomically

- add 1 to the old counter before CAS
- since CAS operates on the whole tagged_index struct we can detect concurrent writes

CAS only succeeds if index AND counter are unchanged.

A concurrent reader can detect the changes due to counter change.

Monotonic counters can be used to detect concurrent changes.

Take with ABA Counter

```
std::optional<T> take() {
  tagged_index newIndex(NO_DATA);
  auto old = m_index.load();

  while (old.index != NO_DATA) {
    newIndex.counter = old.counter + 1; // increment ABA counter
    if (m_index.compare_exchange_strong(old, newIndex)) {
        // we know there was data due to the while loop condition
        auto ret = std::optional<T>(std::move(m_storage[old.index]));
        free(old.index);
        return ret;
    }
    // either retry if CAS failed or exit loop if there is NO_DATA
  }
  return std::nullopt;
}
```

- special case of write where we replace the value with nothing
- need a CAS operation now to increase the counter
- if we just reset the counter during exchange we would still have the ABA problem while reading

Monotonic counters can be used to detect concurrent changes.

Read with ABA Counter

```
std::optional<T> read() {
    auto old = m_index.load();
    while (old.index != NO_DATA) {
      auto ret = std::optional<T>(m_storage[old.index]);

      // check for concurrent changes
      if (m_index.compare_exchange_strong(old, old)) {
        return ret;
      }
      // if this failed either the index or the counter changed
      // (due to a concurrent write)
    }
    return std::nullopt;
  }
```

Same code as before but now with tagged index

Concurrent modifications change the index or the counter

1. concurrent writes
   - index alone may be subject to ABA problem
   - counter increases
2. concurrent take
   - index changes to NO_DATA
   - counter increases
3. concurrent take and writes
   - index alone may be subject to ABA problem
   - counter increases

Monotonic counters can be used to detect concurrent changes.

ABA Problem Solved

	Indexpool<4>				Storage<T, 4>				m_index	
									index	counter
Read at index 1 is interrupted	Free	Used	Free	Free		U U			1 (A)	73
Write X X	Used	Free	Free	Free	X X				0 (B)	74
Write Y Y	Free	Used	Free	Free		Y Y			1 (A)	75
Read at index 1 completes	Free	Used	Free	Free		Y Y			1 (A)	75
Possible read results	Y Y									

Torn read is impossible since the CAS would fail due to counter change.

Replacing std::optional

- std::optional allocates dynamic memory
- allocation is most likely not lock-free

Replace std::optional with a variation of optional that e.g. allocates on the stack.

The copy/move constructor of T also needs to be lock-free.

- this is the case for e.g. trivially copyable types
- memcpy is lock-free, but not atomic

A function is only lock-free if all functions it calls are lock-free.

Restriction to Trivially Copyable Types

- trivially copyable is quite a restriction
- only needed for lock-free reading
- still useful for structs of primitive types, raw memory buffers etc.

Can we drop this restriction?

Yes, but this requires solving the safe memory reclamation problem in a lock-free way …

There are general techniques to solve this problem.

- Hazard Pointers (Maged M. Michael, 2004)
- Key idea: keep track of concurrent users of shared data before reusing the memory
- basically lock-free garbage collection

All these techniques make heavy use of CAS operations as well!

Memory Order

- did not specify memory order explicitly
- implicitly use std::memory_order_seq_cst

Sequentially consistent is the most expensive memory order

- leads to expensive memory barriers

Often sequentially consistent order is not needed

- usually acquire or release order is sufficient
- if we only read/write a variable atomically (no additional synchronization) we can use relaxed memory order

If the algorithm does not work with the most strict order, it generally will not work with weaker orders either.

Optimize memory order once the algorithm is working.

Application in Real-Time Systems

Avoiding Starvation

How useful is this in real-time systems?

- lock-free implementation avoids deadlocks and priority inversion
- progress of at least one contending operation is guaranteed
- operations of individual threads may still starve while in a CAS loop

Solution 1: Add timeouts to CAS loop

- simple to add another termination condition (e.g. timeout) to the loop
- return with a failure indication in case of timeout
- approximate bounds on execution time

Solution 2: Add logic to improve fairness

- add (semi-random) sleeps to allow other threads to make progress
- leads to wait-free algorithms (lock-free and fair)
- overall worse performance

```
bool operation(const T &value) {
  auto deadline = now() +
          std::chrono::milliseconds(100);

  auto old = m_value.load();
  do {
   if (m_value.compare_exchange_strong(old,
                    value)) {
     return true; // success
   }

   // retry or timeout?
   if (now() >= deadline) {
     return false; // timeout
   }
  // … other termination conditions
  } while (true);

  return false;
}
```

Add additional termination conditions to CAS loops to avoid starvation.

Generalizations

We can generalize the techniques we used for the ExchangeBuffer to create lock-free

- Queues (FIFO, Write, Take)
- Stacks (LIFO, Write, Take)
- Caches (FIFO Write, Random Access Read)

ExchangeBuffer can be seen as any of these structures with capacity 1.

Other aspects to consider

1. number of concurrent writers
2. number of concurrent readers
3. size of the internal buffer

Complexity increases due to

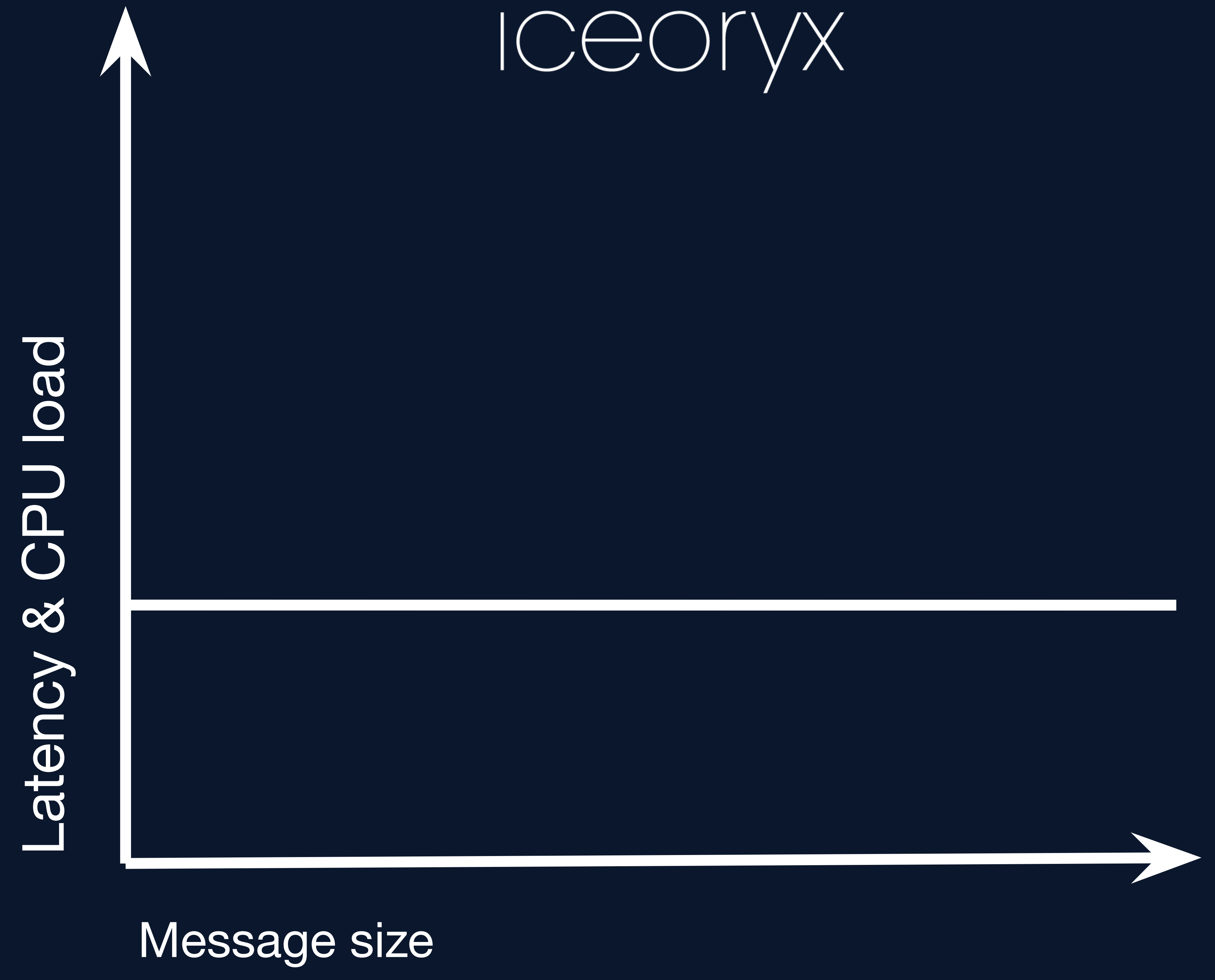
- lock-free management of read- and write-positions in the buffer
- write overflow semantics: e.g. if the queue is full, evict the least recent element and return it
- updating multiple locations atomically without corrupting the structure (e.g. write position and some index)

Applications

The techniques of the ExchangeBuffer are applied in
Eclipse iceoryx ™
https://github.com/eclipse-iceoryx/iceoryx

- zero-copy shared memory middleware
    - data delivered in constant time
- interprocess communication based on lock-free queues (FIFO ringbuffer)
- lock-free shared memory management
- usable in high-safety applications
- suitable for real-time systems due to low latency
- utility library (implements e.g. cxx::optional without dynamic memory)

iceoryx



Testing Lock-free Algorithms

Testing concurrent algorithms is hard - (infinitely) many interleavings of threads

Testing lock-free algorithms is even harder due to concurrent access of data

What can we do to be reasonably sure the algorithm is correct?

1. Define the requirements of the algorithm in a formal way
2. Devise theoretical algorithm (pseudo code)
3. Proof of correctness (by hand or automated theorem prover, e.g. TLA+)
4. Implement algorithm
5. Code review by experts
6. Unit tests to check single-threaded correctness (necessary condition)
7. Multi-threaded stress tests (check invariants, no data structure corruption)
8. Code instrumentation for specific multi-threaded scenarios (open problem …)
9. Verification tools such as CppMem to check for data races etc. (for small code snippets)

In general tests can only disprove correctness of an implementation…
But sufficiently general tests and case coverage increase confidence in correctness.

Our New Friend Compare Exchange

- allows us to implement useful algorithms to exchange data in a lock-free way
- these techniques are especially useful in real-time systems
- care must be taken when manipulating data concurrently without locks

Key Observations

1. CAS loops are the basis for many lock-free algorithms
2. CAS can only be used on small data (64 bit usually)
3. CAS can be used to transfer memory ownership atomically between threads
4. lock-free algorithms usually require managing memory and object lifetime carefully
5. we can add information (e.g. counters) to atomically check for concurrent modification with CAS

Disadvantages

- CAS is rather expensive … but so is locking with contention
- algorithms for simple problems get rather complex
- lock-free algorithms are hard to test

CAS solves the Consensus Problem and can be can be used to synchronize access to concurrent objects for an arbitrary number of threads.

References

iceoryx
https://github.com/eclipse-iceoryx/iceoryx

iceoryx hoofs utility library
- iox::cxx - static memory implementations of e.g. optional
- Iox::cxx::concurrent - lock-free queues

ExchangeBuffer
https://github.com/MatthiasKillat/lockfree_demo
- lock-free implementation (except for std::optional)
- basic unit tests and stress tests
- additionally: basic idea how to update two 64 bit atomic locations consistently (in some sense)
    - this would be easy with Double CAS (not available on standard HW)

C++ Concurrency in Action: Practical Multithreading
Anthony G. Williams

Thanks for the Attention!

Questions?

Devise your own lock-free algorithms!

Replacing std::optional is Not Necessary

Correction of the original talk

The following statements on slide 41 are false:

- std::optional allocates dynamic memory
- allocation is most likely not lock-free

From https://en.cppreference.com/w/cpp/utility/optional

If an optional<T> contains a value, the value is guaranteed to be allocated as part of the optional object footprint, i.e. no dynamic memory allocation ever takes place. Thus, an optional object models an object, not a pointer, even though operator*() and operator->() are defined.

Hence it is not necessary to replace std::optional<T> in the presented interface to obtain a lock-free implementation as there will be no (potentially) blocking dynamic memory allocation due to the optional<T>.

Thanks to Dmytro Dukov for pointing this out.