Concurrent data structures in the ATLAS offline software

Scott Snyder
On behalf of the ATLAS Collaboration

Brookhaven National Laboratory, Upton, NY, USA

November, 2019
CHEP 2019

Introduction

- For Run 3, ATLAS is converting its offline software to run fully multithreaded.
- Some data structures need to be accessed by multiple threads.
- Often accomplished by putting locks around accesses to these structures.
  - Can result in significant overhead for frequently-executed code, even in the absence of contention.
- In come cases, "lockless" methods are known for synchronizing access to data structures without using locking.
  - But are usually complicated and slow for the general case.
  - For few threads, often slower than just using locking.
- Many of the structures of interest to us are "read-mostly."
- Consider allowing multiple lockless readers along with one writer, possibly with locking.
  - Usually much simpler than the general case, with good performance for read-mostly workloads.

CachedValue

Suppose we want to cache ('memoize') the result of a member function (that will always return the same value).

Thread-unsafe example:

```
class Example { ...
  double m_x, m_y;
  mutable double m_r;
  mutable bool m_cached = false;
  double r() const {
    if (!m_cached) {
      m_r = hypot (m_x, m_y);
      m_cached = true;
    }
    return m_r;
  }
```

- Could use std::once, but that uses locks.
- What if we allow the cached function to be possibly evaluated in multiple threads, but set only once?

```
CachedValue<double> m_r;
double r() const {
  if (!m_r.isValid()) {
    m_r.set (hypot (m_x, m_y));
  }
  return m_r.get();
}
```

CachedValue implementation

```
template <class T>
class CachedValue { ...
  enum State { INVALID,
               UPDATING,
               VALID };
  mutable std::atomic<State>
          m_state { INVALID };
  mutable T m_val;
  bool isValid() const {
    State state;
    while ((state = m_state)
           == UPDATING) ;
    return state == VALID;
  }
```

```
void set (T&& val) const {
  State flag = INVALID;
  m_state.compare_exchange_strong
    (flag, UPDATING);
  if (flag == INVALID) {
    m_val = std::move (val);
    m_state = VALID;
  }
  else {
    while (m_state == UPDATING)
      ;
  }
}
```

- Actual code a bit more complicated in order to handle exceptions.
- Simpler, specialized version exists for non-null pointers.

`CachedValue` found to be 2–3× faster than `std::once`.

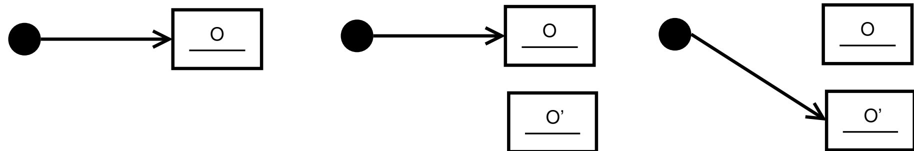
Read-copy-update (RCU)

A family of synchronization strategies useful when

- Reads are frequent and must have low overhead.
- Updates are infrequent and can have larger overhead.
- No strict ordering needed between reads and updates.

Example of such a strategy as used in Athena:

- Have an object referenced via an atomic pointer.
- *Reads* just dereference the pointer (no locking!).
- *Updates* do not change the object in place. Instead, they make an updated copy of the object, then atomically change the pointer from the old object to the new. Updates are serialized wrt other updates, but not wrt reads.

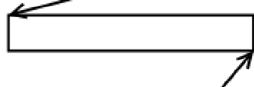


Read-copy-update (RCU)

The “hard” part: deleting the old object. Need to delay deletion until no readers are accessing it any more. Strategy for this is problem-dependent.

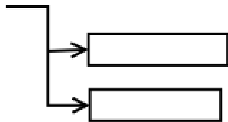
Strategy: Don’t worry about deletion until the owning object is deleted. Example: xAOD cache vector. We have a vector of pointers that we want to access with very low overhead, but also allow it to grow.

atomic<void> m_cache**



atomic<size_t> m_len

vector<void> m_old**



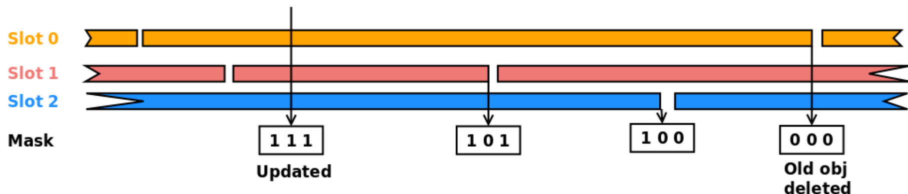
```
void*& Cls::get (size_t ndx) {
  if (ndx >= m_len) {
    lock_t lock (m_mutex);
    if (ndx >= m_len) {
      size_t newlen = m_len*2;
      void** newvec = ...;
      m_old.push_back (m_cache);
      m_cache = newvec;
      m_len = newlen;
    }
  }
  return m_cache[ndx];
```

Read-copy-update (RCU)

The "hard" part: deleting the old object. Need to delay deletion until no readers are accessing it any more. Strategy for this is problem-dependent.

Strategy: Wait until an event has completed for each concurrent slot.

- Object being managed keeps a bitmask for each concurrent slot.
- When object is updated, set bits for all slots.
- When event finishes, clear bit for that slot.
- If all bits are clear, delete accumulated objects.
- Works as long as updates are infrequent.
- Strategy implemented by common Athena RCUSvc.



ConcurrentBitset

Motivation

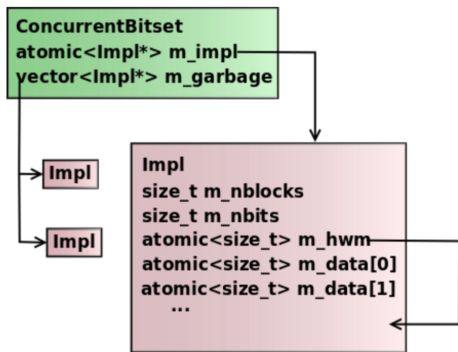
- The ATLAS data model maintains a set of integers for each data object, representing the set of variables available for that object.
- Need to be able to quickly look up elements of the set. Updates are infrequent, but must be thread-safe.

Originally used `unordered_set<unsigned>`, but had high overhead, especially for thread-safety.

Integers are never very big ($< 10^4$), so tried using a bitmap instead.

Represent bitset as an array of `atomic<size_t>`. Allow array to be resized via simple RCU strategy. Allows arbitrary readers concurrent with one writer that may change the array size.

ConcurrentBitset



Some timing comparisons (smaller is better, normalized to 1 for `ConcurrentBitset`).

- `set` and `unordered_set`.
- `concurrent_unordered_set` from TBB.
- `ck_hs` and `ck_bitmap` from ConcurrencyKit.

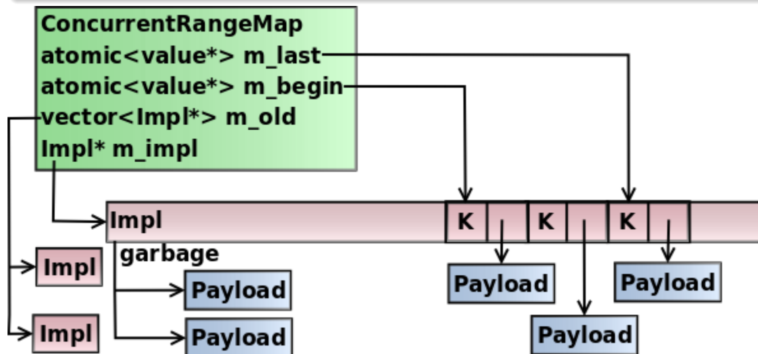
	fill	copy	iterate	lookup
`set`	3.0	11.0	2.4	6.6
`unordered_set`	5.3	15.5	1.5	6.3
`concurrent_unordered_set`	5.8	28.3	2.1	15.0
`ck_hs`	2.5	13.8	4.2	11.0
`ck_bitmap`	0.4	3.5	1.2	0.6
`ConcurrentBitset`	1.0	1.0	1.0	1.0

ConcurrentRangeMap

For time-varying conditions: map from ranges of times to payload objects.

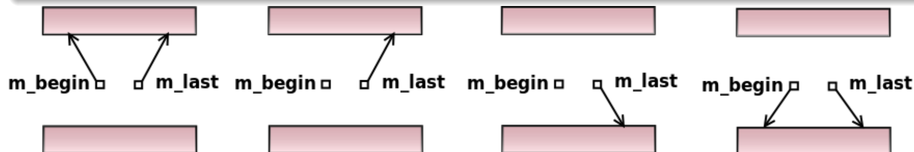
Updates are infrequent, but lookups must be fast.

Use RCU-inspired techniques again. Arbitrary readers can access the structure without locking along with at most one writer (using locking).



ConcurrentRangeMap

When switching to a new new Impl object, both range pointers need to be updated coherently. Need to read/write the pointers carefully so readers will always get a valid range.

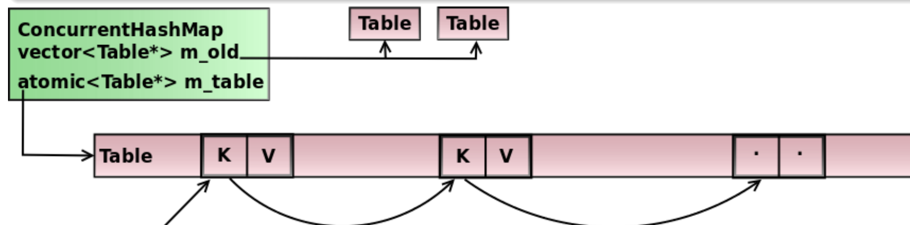



```
// To read the pointers:
do {
  last = m_last;
  begin = m_begin;
} while (!begin || last != m_last);
```

ABA not an issue here since pointers only move in one direction, and if we allocate a new Impl, it will be in a disjoint piece of memory.

Hash Map

If deletions are not needed, a multiple reader/single writer hash table is relatively straightforward, using open addressing.



Need to be able to copy keys atomically. If, e.g., strings are needed, one can allocate them separately and store pointers.

Not currently used in Athena, but initial prototyping is promising, and have several candidate places where such a structure could be useful.

Summary/References

- ATLAS has key data structures that are read frequently but updated rarely.
- Using mutexes to make these thread-safe can lead to significant overhead for reading.
- If only writers are serialized, then readers can often run without locking, yielding good performance for acceptable complexity.

References

- Code: `https://gitlab.cern.ch/ssnyder/concurrentclasses`
- Articles on RCU:
  - `https://lwn.net/Articles/262464`
  - `https://lwn.net/Articles/263130`
  - `https://lwn.net/Articles/573424`
- A number of ideas taken from ConcurrencyKit:
  - `http://concurrencykit.org/`