EPJ Web of Conferences **245**, 05007 (2020) https://doi.org/10.1051/epjconf/202024505007
CHEP 2019

Concurrent data structures in the ATLAS offline software

Scott Snyder[1,*], on behalf of the ATLAS collaboration

[1]Brookhaven National Laboratory, PO Box 5000, Upton NY, 11973, USA

Abstract. In preparation for Run 3 of the LHC, the ATLAS experiment is modifying its offline software to be fully multithreaded. An important part of this is data structures that can be concurrently accessed from many threads both efficiently and safely. A standard way of achieving this is through mutual exclusion; however, the overhead from this can sometimes be excessive. Fully lockless implementations are known for some data structures; however, they are typically complex, and the overhead they require can sometimes be larger than that required for locking implementations. An interesting compromise is to allow lockless access only for reading but not for writing. This often allows the data structures to be much simpler, while still giving good performance for read-mostly access patterns. This proceeding shows some examples of this strategy in data structures used by the ATLAS offline software. It also gives examples of synchronization strategies inspired by read-copy-update, as well as helpers for memoizing values in a multithreaded environment.

1 Introduction

In preparation for Run 3 of the Large Hadron Collider at CERN, the ATLAS experiment [1] is converting its offline software, Athena, to run fully multithreaded [2]. This involves both processing multiple events in different threads (inter-event parallelism) and processing different parts of the same event in different threads (intra-event parallelism). Although one tries to keep each thread as independent as possible, there are some data structures that need to be shared between multiple threads. Accesses to these data structures must be synchronized between threads to prevent data races.

A standard way of doing this is by identifying critical sections of code that require exclusive access to the data structure and put them inside mutual-exclusion (mutex) locks. This works, but if the code is executed frequently and the critical sections are short, this can result in significant overhead due to the locking, even in the absence of any significant contention for the locks.

In some cases, 'lockless' methods are known for synchronizing access to data structures without using explicit locking. But they are usually quite complicated and difficult to make correct. In many cases, they can also be slower than simpler algorithms using explicit locking, as least if there is not much contention for the locks.

However, many of the structures of interest in Athena are 'read-mostly': they are frequently read, and reads are important for performance, but they are only infrequently modified. In this case, one can consider allowing multiple lockless readers along with one writer,

*e-mail: snyder@bnl.gov

where writers may be serialized with each other using locking. This usually turns out to be much simpler than the general case allowing multiple simultaneous writers, while still giving good performance for read-mostly workloads. The remainder of this proceeding will present several examples of this idea as used in the ATLAS offline software.

2 CachedValue

Suppose we have a member function that does some calculation based on member data and returns it, and further suppose that we would like to cache ('memoize') the result. Without taking threading into account, this might be implemented as follows:

```
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;
  }
```

This, however, is not thread-safe.[1] It could be implemented in a thread-safe manner using `std::call_once`, but that typically uses locks. Instead, one could imagine a strategy that separates the calculation of the cached value from storing it. The calculation could possibly be evaluated in multiple threads as long as we ensure that the value is only actually stored to the cache in one thread. The interface for this could look as follows:

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

with an implementation like:

```
template <class T>
class CachedValue { ...
  enum State { INVALID, UPDATING, VALID };
  mutable std::atomic<State> m_state { INVALID };
  mutable T m_val;

  bool isValid() const {
```

[1] Actually, on an x86-family architecture, one could probably get away with this particular example as written. However, it would not be safely portable to other architectures, and it would not work if the value being cached were something more complicated than a single float.

```
  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) ;
  }
}
```

Here, the state of the cached value can be either `INVALID`, meaning it hasn't yet been set, `VALID`, meaning that the value has been set, or `UPDATING`, meaning that some thread is currently updating it. The `set` method uses `compare_exchange_strong` to allow only one thread to change the state from `INVALID` to `UPDATING`, and then to proceed to copy the value and change the state to `VALID`. Both `set` and `isValid` will spin if another thread is writing the value. (The actual implementation is slightly more complicated, to handle the possibility of exceptions among other things. A simplified implementation is also provided for the case where the cached value is a non-null pointer. See the code available at [3].)

Timing tests show that the `CachedValue` implementation here is typically two to three times faster than one using `std::call_once`.[2]

3 Introduction to read-copy-update (RCU)

Read-copy-update (RCU) synchronization [4–7] is a family of synchronization strategies that are broadly applicable in cases where reads are frequent and must have low overhead, updates are infrequent and can have larger overhead, and no strict ordering is required between reads and updates. Such strategies are used in several places in Athena. The typical usage is that one has an object referenced via an atomic pointer. Reads simply dereference the pointer, without performing any locking. Updates do not change the object in place. Rather, they proceed by first making a copy of the object, updating the copy, and then atomically changing the pointer from the old object to the new one. Updates are serialized with other updates, but not with reads.

The old object should eventually be deleted, but one needs to delay doing that until it can be guaranteed that no threads can possibly be still accessing the old object. Exactly how that is done depends on the specific object being considered.

For example, the base container class used in the ATLAS event data model (EDM) [8] contains a vector of pointers used as a cache. This needs to be accessed from multiple threads with very low overhead, but we also need to allow it to grow occasionally. We represent the cache with two atomic variables: a pointer to the start of the cache and its length. If we need to grow the cache, then we allocate a new, longer cache vector, copy the data from the existing one, then update first the pointer variable and then the length. The old cache vector is

[2]The code used for these tests is available as CachedValueTiming_test in [3].

then put on a list to be deleted when the EDM object itself is. As these objects are all deleted after every event, there is no need for more complicated collection strategies. This procedure is outlined below.

```
class xAODBase { ...
  std::atomic<void**> m_cache;
  std::atomic<size_t> m_len;
  std::vector<void**> m_old;
  std::mutex m_mutex;

  void*& get (size_t ndx) {
    if (ndx >= m_len) {
      // Lock against other writers (but not against readers).
      lock_t lock (m_mutex);
      if (ndx >= m_len) {
        size_t newlen = m_len*2;
        void** newvec = ...; // Copy the data.
        m_old.push_back (m_cache);
        // Important to update m_cache before m_len.
        m_cache = newvec;
        m_len = newlen;
      }
    }
    return m_cache[ndx];
  }
```

For objects that persist across many events, a different strategy is needed. Athena assigns each event being processed concurrently to a 'slot', identified by a small integer [2]. The strategy is to wait until an event has completed for each concurrent slot after an update. The object being managed keeps a bit mask for each concurrent slot. When the object is updated, bits for all slots are set. When an event completes, the corresponding bit is cleared in the mask, and when all bits are clear, the object may be deleted, as shown in Figure 1. This strategy works as long as updates are infrequent compared to event processing.

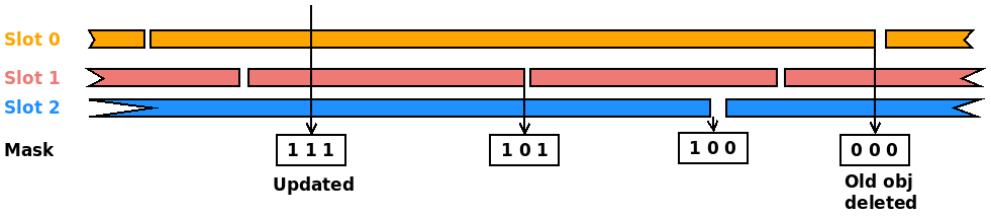

Figure 1. Managing object lifetime with RCU. After an object is updated, the old one can be deleted after each concurrent slot completes processing for an event.

4 ConcurrentBitset

The ATLAS EDM [8] maintains a set of integers for each data object, representing the set of variables available for that object. It is important that lookups in this set be fast. Updates to the set are infrequent, but must be thread-safe. The original implementation used

`std::unordered_set<unsigned>`, but this had a high overhead, especially since, to allow for thread-safe iteration, separate copies of these sets were maintained for each thread. However, the maximum value of these integers is never more than a few thousand, so an alternate implementation in terms of a bit set was explored.

If the bit set is represented as an array of `std::atomic<size_t>`, then concurrent reads and writes can be accommodated naturally. Growing the array is handled with a simple RCU strategy, in which we make a new, larger copy of the array and put the old array on a list to be deleted along with the bit set object itself. This allows for an arbitrary number of readers concurrent with one writer that may change the array size, as shown in Figure 2.

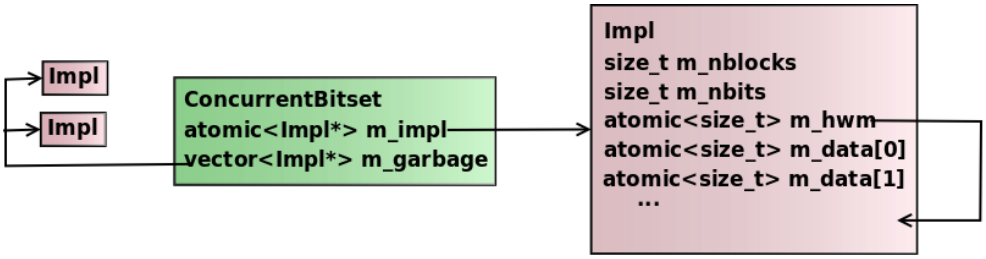

Figure 2. A ConcurrentBitset object points to a (variable-sized) implementation object, which contains an array of words containing the data, along with housekeeping information. If the array needs to grow, a new implementation object is allocated and the old one added to the `m_garbage` list, to be deleted along with the ConcurrentBitset itself.

Table 1 shows some (single-threaded) timing comparisons between ConcurrentBitset and several alternate implementations, including `set` and `unordered_set` from the C++ library, `concurrent_unordered_set` from Intel Threading Building Blocks [9], and `ck_hs` (hash set) and `ck_bitmap` from ConcurrencyKit [10], for several representative operations that are important for the Athena EDM use case. Overall, ConcurrentBitset performs quite well on these tests.

Table 1. Single-threaded timing comparisons between ConcurrentBitset and alternate implementations. Smaller is better, and each (dimensionless) column is normalized to set ConcurrentBitset to 1.0. (Based on running ConcurrentBitset_test from [3] with the `-perf` option.)

	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

5 ConcurrentRangeMap

To handle time-varying conditions data [11], Athena maintains a mapping for each data item from ranges of times to payload objects. Lookups in these maps must be fast, but updates are relatively infrequent, so RCU-inspired techniques are again used.

The map is laid out as a linear vector of pairs of time ranges and pointers to payload objects, as shown in Figure 3. Two atomic pointers point at the first and the last valid entry in the vector, respectively. Lookups proceed without performing any locking. The ranges are kept in sorted order; however, we expect that the most recent (largest) range will be the one accessed most frequently. Therefore, lookups proceed by first testing the last valid entry, and if that fails, by then performing a binary search among the remaining entries.

Updates to the map are serialized with other updates, but not with reads. Removing the first valid entry can be done by simply incrementing the atomic begin pointer; the payload object is then added to a list to be deleted when the vector itself is. Adding a new entry to the end is also simply done by copying the new time range and payload pointer to the vector and then (following a `std::atomic_thread_fence`) incrementing the atomic last pointer.

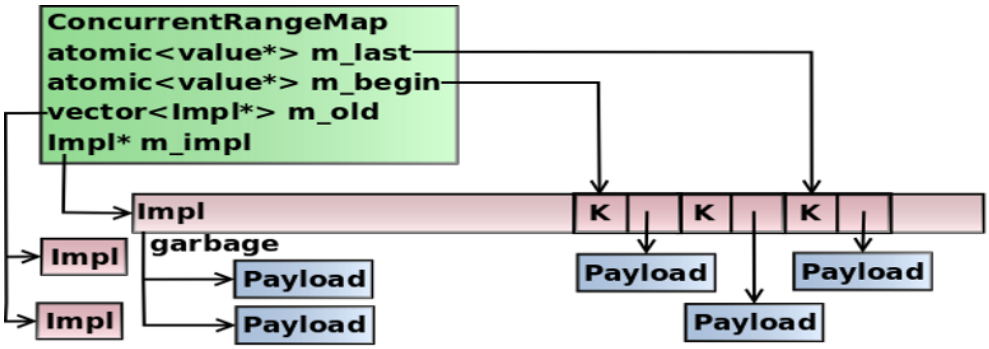

Figure 3. A ConcurrentRangeMap consists of a variable-sized implementation object containing a vector of pairs of time ranges and pointers to payload objects. Two atomic pointers delimit the range of valid entries. Old implementation objects are kept on a list to be deleted once no thread can any more be accessing them. Payload objects from deleted entries are put on a list in the implementation object to be deleted along with it.

If there is no more room at the end of the vector, or if insertions or deletions from other positions in the vector are required, then a new vector is allocated and the valid entries are copied to the start of the new vector. The atomic pointers are adjusted to point to the new vector, and the old one is placed on a list to be deleted when it is no longer in use (using the second of the RCU strategies discussed above).

Adjusting the two atomic pointers that delimit the valid entries requires some care, in order to ensure that readers will always see a consistent range of valid entries. The pointers are adjusted by first setting the begin pointer to null, then updating the end pointer, and finally by updating the begin pointer, as shown in Figure 4. The pointers are then read using the following sequence:

```
do {
  // The order of these two reads is important!
  last = m_last;
  begin = m_begin;
} while (!begin || last != m_last);
```

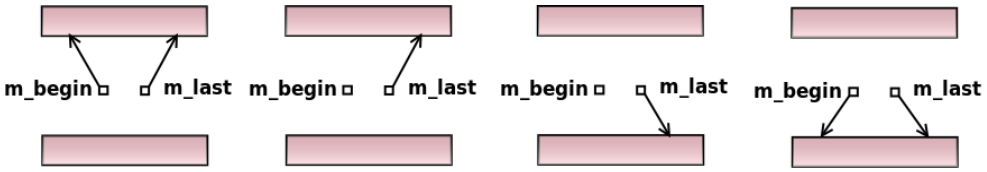

Figure 4. Procedure for updating the two pointers that delimit the valid entries (see text).

6 Summary

This proceeding describes several examples of concurrent data structures used by the ATLAS offline code, intended for read-mostly workloads, that support multiple lockless readers along with a single serialized writer. The corresponding code for the classes described here is available at [3].

Acknowledgments

This work is supported in part by the U.S. Department of Energy under contract DE-AC02-98CH10886 with Brookhaven National Laboratory. The author would like to acknowledge ConcurrencyKit [10] for a number of ideas related to the implementation of the classes described here.

References

[1] ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST **3**, S08003 (2008), http://doi.org/10.1088/1748-0221/3/08/S08003

[2] S. Kama, C. Leggett, S. Snyder, V. Tsulaia, *The ATLAS multithreaded offline framework*, EPJ Web Conf. **214**, 05018 (2019), https://doi.org/10.1051/epjconf/201921405018

[3] https://gitlab.cern.ch/ssnyder/concurrentclasses [accessed 2020-01-14]

[4] P.E. McKenney, J. Walpole, *What is RCU, Fundamentally?*, Linux Weekly News (2007), https://lwn.net/Articles/262464/ [accessed 2020-01-13]

[5] P.E. McKenney, J. Walpole, *What is RCU? Part2: Usage*, Linux Weekly News (2007), https://lwn.net/Articles/263130/ [accessed 2020-01-13]

[6] P.E. McKenney, M. Desnoyers, L. Jiangshan, *User-space RCU*, Linux Weekly News (2013), https://lwn.net/Articles/573424/ [accessed 2020-01-13]

[7] M. Desnoyers, P.E. McKenney, A.S. Stern, M.R. Dagenais, J. Walpole, *User-level implementations of read-copy update*, IEEE Transactions on Parallel and Distributed Systems **23**, 375 (2012), https://ieeexplore.ieee.org/document/5871597

[8] A. Buckley, T. Eifert, M. Elsing, D. Gillberg, K. Koeneke, A. Krasznahorkay, E. Moyse, M. Nowak, S. Snyder, P. van Gemmeren, *Implementation of the ATLAS Run 2 event data model*, J. Phys. Conf. Ser. **664**, 072045 (2015), https://doi.org/10.1088/1742-6596/664/7/072045

[9] *Intel Threading Building Blocks*, https://github.com/intel/tbb [accessed 2020-01-14]

[10] *ConcurrencyKit*, http://concurrencykit.org/ [accessed 2020-01-14]

[11] C. Leggett, I. Shapoval, S. Snyder, V. Tsulaia, *Conditions DataHandling in the Multithreaded ATLAS Framework*, EPJ Web Conf. **214**, 05031 (2019), https://doi.org/10.1051/epjconf/201921405031