Atomic minimum/maximum

Document #: D0493R3 **DRAFT**
Date: 2021-12-02
Project: Programming Language C++
Audience: WG21 SG21 (Contracts)
Reply-to: Al Grant
<al.grant@arm.com>
Bronek Kozicki
<brok@spamcop.net>
Tim Northover
<tnorthover@apple.com>

Contents

1 Abstract

Add integer *max* and *min* operations to the set of operations supported in `<atomic>`. There are minor adjustments to function naming necessitated by the fact that `max` and `min` do not exist as infix operators.

2 Changelog

— Revision R3, published 2021-12-??
  — Changed formatting
  — Remove an exceedingly long motivating example
  — Following LEWG feedback, revert to require store if the value does not change
  — Rebase on draft [N4901] **TODO**
  — Add floating numbers support to wording, consistently with [P0020] **TODO**

  - Modernize remaining motivating example
  - Add example implementation based on CAS loop
- Revision R2, published 2021-05-11
  - Change proposal to make the store unspecified if the value does not change
  - Align with C++20
- Revision R1, published 2020-05-08
  - Add motivation for defining new atomics as read-modify-write
  - Clarify status of proposal for new-value-returning operations.
  - Align with C++17.
- Revision R0 pulished 2016-11-08
  - Original proposal

3 Introduction

This proposal extends the atomic operations library to add atomic maximum/minimum operations. These were originally proposed for C++ in [N3696] as particular cases of a general "priority update" mechanism, which atomically combined reading an object's value, computing a new value and conditionally writing this value if it differs from the old value.

In revision R2 of this paper we have proposed atomic maximum/minimum operations where it is unspecified whether or not the store takes place if the new value happens to be the same as the old value. This has caused contention in LEWG, for example:

> The idea of leaving whether a write occurs or not unspecified has no precedent and doesn't play well with memory ordering; it's not possible to have `memory_order_release` (or `memory_order_acq_rel`) without a write. (…) I see several ways out of this, but think that only one of them makes sense: guarantee *read-modify-write* behavior if the memory order contains release.

The primary purpose of R3 is to propose a solution which is discussed in section 5.

4 Background and motivation

Atomic addition (*fetch-and-add*) was introduced in the NYU Ultracomputer [Gottlieb 1982], has been implemented in a variety of hardware architectures, and has been standardized in C and C++. Atomic maximum/minimum operations (*fetch-and-max* , *fetch-and-min*) have a history almost as long as atomic addition, e.g. see [Lipovski 1988], and have also been implemented in various hardware architectures but are not currently standard in C and C++. This proposal fills the gap.

Atomic maximum/minimum operations are useful in a variety of situations in multithreaded applications:

- optimal implementation of lock-free shared data structures - as in the motivating example later in this paper
- reductions in data-parallel applications: for example, OpenMP supports maximum as a reduction operation
- recording the maximum so far reached in an optimization process, to allow unproductive threads to terminate
- collecting statistics, such as the largest item of input encountered by any worker thread.

Atomic maximum/minimum operations already exist in several other programming environments, including OpenCL, and in some hardware implementations. Application need, and availability, motivate providing these operations in C++.

The proposed language changes add atomic max/min to `<atomic>`, with some syntatic adjustment due to the fact that C++ has no infix operators for max/min.

5 The problem of conditional write

The existing atomic operations (e.g. `fetch_and`) have the effect of a *read-modify-write*, irrespective of whether the value changes. This is how atomic max/min are defined in several APIs (OpenCL, CUDA, C++AMP, HCC) and in several hardware architectures (ARM, RISC-V). However, some hardware (POWER) implements atomic max/min as an atomic *read-and-conditional-store*.

If we look at an example CAS-loop implementation of this proposal, it is easy to see why such *read-and-conditional-store* can be potentially more performant.

5.1 Example CAS-loop implementation with *read-modify-write*

In this version we are performing unconditional store, which means all writers need exclusive cache line access. This may result in excessive writer contention.

```
template <typename T>
T atomic_fetch_max_explicit(atomic<T>* pv,
                            typename atomic<T>::value_type v,
                            memory_order m) noexcept {
    auto t = pv->load(memory_order_acquire);
    while (!pv->compare_exchange_weak(t, max(v, t), m, memory_order_relaxed))
      ;
    return t;
}
```

5.2 Example CAS-loop implementation with *read-and-conditional-store*

Please note in this version the condition of the `while` loop, which will cause the algorithm to skip write entirely if the atomic is already equal to the post-condition `max(v, t)`. This may significantly reduce writer contention, since up to this point the algorithm only requires shared cache line access.

```
template <typename T>
T atomic_fetch_max_explicit(atomic<T>* pv,
                            typename atomic<T>::value_type v,
                            memory_order m) noexcept {
    auto t = pv->load(memory_order_acquire);
    while (max(v, t) != t) {
        if (pv->compare_exchange_weak(t, v, m, memory_order_relaxed))
            return t;
    }
    return t;
}
```

If we require store always, then the CAS algorithm like the one presented above would be nonconforming, even though it is potentially more efficient. Similarly a hardware instruction which exhibits behaviour similar to the above algorithm would be non-conforming.

We would like to "eat cake and have it", more specifically:

— implement conforming atomic min/max on hardware which might not perform stores, without falling back to CAS algorithm
— expose a potentially more performant implementation when e.g. a memory order requested by the user does not contain release

We think this might be possible, but are uncertain as to the additional wording required (if any), as demonstrated below.

5.3 Example CAS-loop implementation with *read-modify-write* (synthetic)

The idea is to allow for a "synthetic" write at the same point where the preceding algorighm performed `load()`.

```
template <typename T>
T atomic_fetch_max_explicit(atomic<T>* pv,
                            typename atomic<T>::value_type v,
                            memory_order m) noexcept {
    auto t = ((m == memory_order_relaxed || m == memory_order_acquire)
              ? pv->load(m)
              : pv->fetch_add(0, m));
    while (max(v, t) != t) {
        if (pv->compare_exchange_weak(t, v, m, memory_order_relaxed))
            return t;
    }
    return t;
}
```

The `fetch_add` operation added above ensures that the (half-)barrier `m` is applied at the program point expected by the user. On architectures which implement atomic min/max in hardware with *read-and-conditional-store* semantics, this extra store could be performed with a preceeding instruction hence allowing conforming implementation which does not have to perform a CAS loop. Such an implementation might also perform a conditional check similar to demonstrated above, hence exposing a potentially more performant implementation if no release was requested by the user (the authors are unsure about consume memory order).

Reader will also notice that the above algorithm does not contain hardcoded and arbitrary memory order `memory_order_acquire` - instead we rely on memory order requested by the user, which seem more correct.

5.4 The wording problem

Assuming the proposed solution meets approval, the authors are unsure what (if any) wording should be used to define the semantics of such atomics. Is it possible that the proposed solution meets current requirements of *read-modify-write* operations as currently specified in the standard? That would be an ideal situation, but is probably too good to be true.

6 Motivating example

Atomic fetch-and-max can be used to implement a lockfree bounded multi-consumer, multi-producer queue. Below is an example based on [Gong 1990]. Note, the original paper assumed existence of `EXCHANGE` operation which in practice does not exist on many platforms. Here this was replaced by a two-step read and write, in addition to translation from C to C++. For this reason the correctness proof from [Gong 1990] does not apply.

```
template <typename T, size_t Size>
struct queue_t {
  using elt = T;
  static constexpr int size = Size;

  struct entry {
    elt item {};                          // a queue element
    std::atomic<int> tag {-1};            // its generation number
  };

  entry elts[size] = {};                  // a bounded array
  std::atomic<int> back {-1};

  friend void enqueue(queue_t& queue, elt x) noexcept {
```

```
    int i = queue.back.load() + 1;          // get a slot in the array for the new element
    while (true) {
      // exchange the new element with slots value if that slot has not been used
      int empty = -1;                       // expected tag for an empty slot
      auto& e = queue.elts[i % size];
      // use two-step write: first store an odd value while we are writing the new element
      if (std::atomic_compare_exchange_strong(&e.tag, &empty, (i / size) * 2 + 1)) {
        using std::swap;
        swap(x, e.item);
        e.tag.store((i / size) * 2);        // done writing, switch tag to even (ie. ready)
        break;
      }
      ++i;
    }
    std::atomic_fetch_max(&queue.back, i);  // reset the value of back
  }

  friend auto dequeue(queue_t& queue) noexcept -> elt {
    while (true) {                          // keep trying until an element is found
      int range = queue.back.load();        // search up to back slots
      for (int i = 0; i <= range; i++) {
        int ready = (i / size) * 2;         // expected even tag for ready slot
        auto& e = queue.elts[i % size];
        // use two-step read: first store -2 while we are reading the element
        if (std::atomic_compare_exchange_strong(&e.tag, &ready, -2)) {
          using std::swap;
          elt ret{};
          swap(ret, e.item);
          e.tag.store(-1);                  // done reading, switch tag to -1 (ie. empty)
          return ret;
        }
      }
    }
  }
};
```

7 Summary of proposed additions to <atomic>

The current <atomic> provides atomic operations in several ways:

— as a named non-member function template e.g. atomic_fetch_add returning the old value
— as a named member function template e.g. atomic<T>::fetch_add() returning the old value
— as an overloaded compound operator e.g. atomic<T>::operator+=() returning the new value

Adding 'max' and 'min' versions of the named functions is straightforward. Unlike the existing atomics, max/min operations exist in signed and unsigned flavors. The atomic type determines the operation. There is precedent for this in C, where all compound assignments on atomic variables are defined to be atomic, including sign-sensitive operations such as divide and right-shift.

The overloaded operator atomic<T>::operator*key*=(n) is defined to return the new value of the atomic object. This does not correspond directly to a named function. For max and min, we have no infix operators to overload. So if we want a function that returns the new value we would need to provide it as a named function. However, for all operators the new value can be obtained as fetch_*key*(n)*key*n, (the standard defines the compound operator overloads this way) while the reverse is not true for non-invertible operators like 'and' or 'max'. Thus

the functions would add no significant functionality other than providing one-to-one equivalents to `<atomic>` existing compound operator overloads. Following some of the early literature on atomic operations ([Kruskal 1986] citing [Draughon 1967]), we suggest that these names should have the form `replace_`*key*, as shown in the wording section.

The authors are happy to remove these `replace_` functions from the paper, if such feedback is received. They are provided in this version of the paper for the sake of interface completness.

8 Implementation experience

The required intrinsics have been added to Clang.

9 Acknowlegments

This paper benefited from discussion with Mario Torrecillas Rodriguez, Nigel Stephens and Nick Maclaren.

10 Changes to the C++ standard

Proposed changes are missing floating point support, which will to be added before publication

As discussed above, the write semantics of proposed atomic min/max may require additional wording

The following text outlines the proposed changes, based on [N4868].

31: Atomic operations library [atomics]

31.2: Header synopsis [atomics.syn]

Add:

```
namespace std {
  // 31.9, non-member functions
  ...
  template<class T>
    T atomic_fetch_max(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
  template<class T>
    T atomic_fetch_max(atomic<T>*, typename atomic<T>::value_type) noexcept;
  template<class T>
    T atomic_fetch_max_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexce
  template<class T>
    T atomic_fetch_max_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
  template<class T>
    T atomic_fetch_min(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
  template<class T>
    T atomic_fetch_min(atomic<T>*, typename atomic<T>::value_type) noexcept;
  template<class T>
    T atomic_fetch_min_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexce
  template<class T>
    T atomic_fetch_min_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
  ...
}
```

31.7.3: Specializations for integral types [atomics.ref.int]

Add:

```
namespace std {
  template <> struct atomic_ref<integral> {
    ...
    integral fetch_max(integral, memory_order = memory_order_seq_cst) const noexcept;
    integral fetch_min(integral, memory_order = memory_order_seq_cst) const noexcept;
    ...
    integral replace_max(integral) const noexcept;
    integral replace_min(integral) const noexcept;
  };
}
```

Change:

Remarks: For signed integer types, the result is as if the object value and parameters were converted to their corresponding unsigned types, the computation performed on those types, and the result converted back to the signed type.

to:

Remarks: Except for `fetch_max` and `fetch_min`, for signed integer types, the result is as if the object value and parameters were converted to their corresponding unsigned types, the computation performed on those types, and the result converted back to the signed type. For `fetch_max` and `fetch_min`, the computation is performed according to the integral type of the atomic object.

Add the following text:

`integral A::replace_`*key*`(integral operand) const noexcept;`

Requires: These operations are only defined for keys `max` and `min`.

Effects: `A::fetch_`*key*`(operand)`

Returns: `std::`*key*`(A::fetch_`*key*`(operand), operand)`

After `integral operatorop=(integral operand) const noexcept;` add:

These operations are not defined for keys `max` and `min`.

31.7.5: Partial specialization for pointers [atomics.ref.pointer]

```
namespace std {
  template <class T> struct atomic_ref<T *> {
    ...
    T* fetch_max(T *, memory_order = memory_order::seq_cst) const noexcept;
    T* fetch_min(T *, memory_order = memory_order::seq_cst) const noexcept;
  };
}
```

Add the following text:

`T* A::replace_`*key*‘(T* operand) const noexcept;

Requires: These operations are only defined for keys `max` and `min`.

Effects: `A::fetch_`*key*`(operand)`

Returns: `std::`*key*`(A::fetch_`*key*`(operand), operand)`

31.8.3: Specializations for integers [atomics.types.int]

```
namespace std {
  template <> struct atomic<integral> {
    ...
```

```
    integral fetch_max(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral fetch_max(integral, memory_order = memory_order_seq_cst) noexcept;
    integral fetch_min(integral, memory_order = memory_order_seq_cst) volatile noexcept;
    integral fetch_min(integral, memory_order = memory_order_seq_cst) noexcept;
    ...
  };
}
```

In table 144, [tab:atomic.types.int.comp], add the following entries:

Key	Op	Computation
max		maximum as computed by `std::max` from `<algorithm>`
min		minimum as computed by `std::min` from `<algorithm>`

Add the following text:

`C A::replace_`*key*`(M operand) volatile noexcept;`

`C A::replace_`*key*`(M operand) noexcept;`

Requires: These operations are only defined for keys `max` and `min`.

Effects: `A::fetch_`*key*`(operand)`

Returns: `std::`*key*`(A::fetch_`*key*`(operand), operand)`

After `T* operatorop=(T operand) noexcept;` add:

These operations are not defined for keys `max` and `min`.

31.8.5: Partial specialization for pointers [atomics.types.pointer]

```
namespace std {
  template <class T> struct atomic<T*> {
    ...
    T* fetch_max(T*, memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_max(T*, memory_order = memory_order_seq_cst) noexcept;
    T* fetch_min(T*, memory_order = memory_order_seq_cst) volatile noexcept;
    T* fetch_min(T*, memory_order = memory_order_seq_cst) noexcept;
    ...
  };
}
```

In table 145, [tab:atomic.types.pointer.comp], add the following entries:

Key	Op	Computation
max		maximum as computed by `std::max` from `<algorithm>`
min		minimum as computed by `std::min` from `<algorithm>`

Add:

`C A::replace_`*key*`(M operand) volatile noexcept;`

`C A::replace_`*key*`(M operand) noexcept;`

Requires: These operations are only defined for keys `max` and `min`.

Effects: `A::fetch_`*key*`(operand)`

Returns: `std::`*key*`(A::fetch_`*key*`(operand), operand)`

After `T* operatorop=(T operand) noexcept;` add:

These operations are not defined for keys `max` and `min`

11 References

[Draughon 1967] E. Draughon, Ralph Grishman, J. Schwartz, and A. Stein. Programming Considerations for Parallel Computers.
https://nyuscholars.nyu.edu/en/publications/programming-considerations-for-parallel-computers

[Gong 1990] Chun Gong and Jeanette M. Wing. A Library of Concurrent Objects and Their Proofs of Correctness.
http://www.cs.cmu.edu/~wing/publications/CMU-CS-90-151.pdf

[Gottlieb 1982] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer.
https://ieeexplore.ieee.org/document/1676201

[Kruskal 1986] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient Synchronization on Multiprocessors with Shared Memory.
https://dl.acm.org/doi/10.1145/48022.48024

[Lipovski 1988] G. J. Lipovski and Paul Vaughan. A Fetch-And-Op Implementation for Parallel Computers.
https://ieeexplore.ieee.org/document/5249

[N3696] Bronek Kozicki. 2013-06-26. Proposal to extend atomic with priority update functions.
https://wg21.link/n3696

[N4868] Richard Smith. 2020-10-18. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4868

[N4901] Thomas Köppe. 2021-10-22. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4901

[P0020] H. Carter Edwards, Hans Boehm, Olivier Giroux, JF Bastien, and James Reus. P0020r6 : Floating Point Atomic.
https://wg21.link/p0020r6