Atomic minimum/maximum

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 R5, published 2024-02-15

— In wording, drop changes in sections `[atomics.types.float]` and `[atomics.ref.float]`
— Add note on floating point operations
— Improve example polyfill implementation, move it to separate section 5.3
— Add implementation note
— Update benchmarks for improved polyfill
— Revision R4, published 2022-11-15
  — Drop unusable benchmark
  — Rebase on draft [N4917]
  — Add "freestanding" to the wording of non-member functions
  — In wording, add remarks to explain `fetch_max` and `fetch_min` operations
  — In wording, add note on requirements of comparing pointers
  — Add note on pointer operations
— Revision R3, published 2021-12-15
  — Change formatting
  — Revert to *read-modify-write* semantics, based on SG1 feedback
  — Remove `replace_`*key* functions, based on SG1 feedback
  — Simplify wording
  — Add floating numbers support to wording
  — Add feature test macro
  — Remove one (exceedingly long) motivating example
  — Rewrite other motivating example in modern C++
  — Rebase on draft [N4901]
  — Add example implementation based on CAS loop
  — Add benchmark comparing hardware vs CAS-loop implementation
— 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, but upon further discussion in SG1 turned out to be unnecessary - as 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 in C++.

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>` for builtin types, including integral, pointer and floating point.

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 more efficient.

Following the discussion in SG1 the authors are convinced that a similar implementation can be conforming, *with some adjustments* (example presented in 5.3), without the catch all wording such as "*it is unspecified whether or not the store takes place*".

Note

Example polyfill implementations listed below rely on a simple helper function whose task is to adjust `memory_order` to make it a valid operand for the load operations:

```
constexpr inline memory_order drop_release(memory_order m) noexcept {
  return (m == memory_order_release ? memory_order_relaxed
      : ((m == memory_order_acq_rel || m == memory_order_seq_cst) ? memory_order_acquire
          : m));
}
```

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

This implementation skips writes entirely if `pv` is already equal to `max(v, t)`. It **does not** conform with the *read-modify-write* semantics, which this paper proposes:

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

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

This implementation is performing an unconditional store, which means all writers need exclusive cache line access. Although conforming with the *read-modify-write* semantics, this may result in excessive writer contention:

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

5.3 Example improved CAS-loop implementation with *read-modify-write*

This implementation is based on *read-and-conditional-store*, with an added extra step to ensure that a store does take place at least once, *if required*:

— if the user requested memory order is *not* a release, then store is not required
— otherwise, add a dummy write such as `fetch_add(0, m)` and use its result.

This is demonstrated below:

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

    return t;
}
```

A subtle difference between this and the previous implementation is that, in this case, an extra "dummy" store can take place. The authors argue that this difference in behaviour is unobservable in the standard C++ memory model.

Similarly, given an architecture which implements atomic minimum/maximum in hardware with *read-and-conditional-store* semantics, a conforming *read-modify-write* `fetch_max()` can be implemented with very little overhead.

For this reason **and** for consistency with all other atomic instructions, we have decided to use *read-modify-write* semantics for the proposed atomic minimum/maximum.

6 Infix operators in `<atomic>` and min/max

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 new functions returning the new result would add no significant functionality other than providing one-to-one equivalents to `<atomic>` existing compound operator overloads. Revision R2 of this paper tentatively suggested such functions, named `replace_`*key* (following some of the early literature on atomic operations - [Kruskal 1986] citing [Draughon 1967]). Having discussed this in SG1, the authors have decided *not* to propose addition of extra functions and correspondingly they have been *removed* in revision R3. This same result can be obtained by the user with a simple expression such as `max(v.fetch_max(x), x)` or `min(v.fetch_min(x), x)`.

During the discussion in SG1, it was suggested that a new paper could be written proposing *key*`_fetch` functions returning **new** values. This is *not* such paper.

7   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 most 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.

```cpp
template <typename T, size_t Size>
struct queue_t {
  static_assert(std::is_nothrow_default_constructible_v<T>);
  static_assert(std::is_nothrow_copy_constructible_v<T>);
  static_assert(std::is_nothrow_swappable_v<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;
      }
    }
  }
}
};
```

8   Implementation experience

The required intrinsics have been added to Clang.

9   Benchmarks

We have implemented benchmark `bench` and made it available on [Github].

— `bench` is finding a maximum value from a PRNG. We were able to achieve acceptably low standard deviation of results for this test. The selected PRNG is a linear distribution 2e9 wide, using 10'000 PRNG samples per run. In this benchmark, the `fetch_max` updates were relatively infrequent.

We have measured the nanosecond time of different implementations of:

```
atomic_fetch_max_explicit(&max, i, std::memory_order_acq_rel)
```

where `i` is generated by a PRNG. The benchmarks capture the cost of contention to `max` from varying number of cores. The benchmarks were run on AWS EC2 instance type `c7g.16xlarge` (i.e. 64 cores ARMv8.4 Graviton3 CPU). The machine was running Linux kernel 6.1 and was configured for complete isolation of cores 1-63. We used core 0 only when running the benchmark across all 64 cores, in which case the samples from this core were dropped (to avoid the noise caused by the normal operating system operation).

The benchmark parameters were:

— `-m 0.5` : maximum std. deviation for PRNG cost calibration
— `-i 1e6` : number of iterations (this translates to 100 runs, each sampling the PRNG 10'000 times)

The table below compares two `fetch_max` implementations:

— `-t t` : CAS-loop based algorithm presented in 5.3 (we call it "smart")

— `-t h` : hardware instruction `ldsmaxl` available in ARM8.1 instruction set

	CAS-loop "smart"			Hardware instruction	
Cores	Time ns	Std. deviation		Time ns	Std. deviation
2	13	1		13	1
4	23	2		22	2
8	74	12		74	13
16	258	22		238	22
24	443	28		406	22
32	634	34		586	27
40	834	39		772	34
48	1005	42		942	37
56	1194	47		1114	39
64	1380	48		1294	46

During benchmarking, we have observed that the time of *read-and-conditional-store* CAS-loop algorithm (as presented in 5.2, we call it "weak" in the benchmarks) was almost immeasurable, *irrespective* of the number of cores. We explain this by how rarely the PRNG sampling benchmark updates the `max` value. Similarly the "smart" algorithm with `std::memory_order_release` had shown to be very fast, *irrespective* of the number of cores.

10 Note on pointer operations

It was pointed out that the semantics of pointer operations is not clear in the revision 3 of this paper. The new wording in revision 4 makes it clear that the new atomic operations perform computation as-if by `max` and `min` algorithms, which also work on pointers if these point the same complete object (or array), see [expr.rel] remark 4. The intent is to give `fetch_max` and `fetch_min` the same semantics, including the requirements.

This is in apparent divergence from other atomic operations which are guaranteed not to create undefined behaviour (e.g. *"If the result is not a representable value for its type (7.1), the result is unspecified, but the operations otherwise have no undefined behavior."* for floating point and *"The result may be an undefined address, but the operations otherwise have no undefined behavior."* for pointers). Note that `fetch_max` and `fetch_min` *in principle* do not create new values as opposed to other atomic functions; the result of the function is either an old value of the atomic object or a new value, as provided by the caller. Hence there's less demand for an "escape clause" for potentially "undefined address" (there *likely* isn't one).

If this proposal is accepted and we gain more experience with existing implementations of `fetch_max` and `fetch_min`, plausibly an "escape clause" similar to ones quoted above *might* be added in the future revisions of C++ e.g. by allowing comparison of unrelated pointers. At this moment we aren't certain that such hypothetical clause would be implementable; furthermore a user with a need for such operation could use conversion to and from `uintptr_t` instead (and deal with the fallout of using resulting pointer).

11 Note on floating point operations

Following the discussion in Varna '23 plenary, also carried on the reflector, the authors decided to remove the proposed `fetch_min` and `fetch_max` from the loating point specializations (that is, sections [atomics.types.float] and [atomics.ref.float]).

Floating point types do not receive the same treatment in `std::min` and `std::max` as other types do (due to the presence of NaN values and signed zero), hence they would have to be either defined using different means, or at the very least worded differently. Since there already is implementation experience regarding the use of `std::fmin` and `std::fmax` in atomic operations on floating point numbers, and a new paper [P3008] is being

prepared to propose the relevant addition in the standard, trying to nail down the semantics based on `std::min` and `std::max` in this proposal seems counterproductive.

12   Acknowlegments

13   Changes to the C++ standard

The following text outlines the proposed changes, based on [N4917].

17 Language support library [support]

17.3 Implementation properties [support.limits]

17.3.2 Header `<version>` synopsis

Add feature test macro:

```
#define __cpp_lib_atomic_min_max  202XXXL // also in <atomic>
```

33 Concurrency support library [thread]

33.5 Atomic operations [atomics]

33.5.2 Header `<atomic>` synopsis [atomics.syn]

   — *Add following functions, immediately below `atomic_fetch_xor_explicit`:*

```
namespace std {
  // 33.5.9, non-member functions
  ...
  template<class T>
    T atomic_fetch_max(volatile atomic<T>*,                   // freestanding
                       typename atomic<T>::value_type) noexcept;
  template<class T>
    T atomic_fetch_max(atomic<T>*,                            // freestanding
                       typename atomic<T>::value_type) noexcept;
  template<class T>
    T atomic_fetch_max_explicit(volatile atomic<T>*,          // freestanding
                                typename atomic<T>::value_type,
                                memory_order) noexcept;
  template<class T>
    T atomic_fetch_max_explicit(atomic<T>*,                   // freestanding
                                typename atomic<T>::value_type,
                                memory_order) noexcept;
  template<class T>
    T atomic_fetch_min(volatile atomic<T>*,                   // freestanding
                       typename atomic<T>::value_type) noexcept;
  template<class T>
    T atomic_fetch_min(atomic<T>*,                            // freestanding
                       typename atomic<T>::value_type) noexcept;
  template<class T>
    T atomic_fetch_min_explicit(volatile atomic<T>*,          // freestanding
                                typename atomic<T>::value_type,
                                memory_order) noexcept;
  template<class T>
```

```
    T atomic_fetch_min_explicit(atomic<T>*,                          // freestanding
                                typename atomic<T>::value_type,
                                memory_order) noexcept;
  ...
}
```

33.5.7 Class template atomic_ref [atomics.ref.generic]

33.5.7.3 Specializations for integral types [atomics.ref.int]

— *Add following public functions, immediately below* `fetch_xor`:

```
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;
    ...
  };
}
```

— *Change remark 6:*

6   *Remarks*: ~~For~~ 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.

       [*Note 2 : There are no undefined results arising from the computation. —- end note*]

— *Add remark 7 immediately below:*

7   *Remarks*: For `fetch_max` and `fetch_min`, the maximum and minimum computation is performed as if by
       `max` and `min` algorithms [alg.min.max], respectively, with the object value and the first parameter as the
       arguments.

— *Bump existing remarks below new remark 7*

33.5.7.5 Partial specialization for pointers [atomics.ref.pointer]

— *Add following public functions, immediately below* `fetch_sub`:

```
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 remark 7 with note 1 immediately below remark 6:*

7   *Remarks*: For `fetch_max` and `fetch_min`, the maximum and minimum computation is performed as if by
       `max` and `min` algorithms [alg.min.max], respectively, with the object value and the first parameter as the
       arguments.

       [*Note 1: If the pointers point to different complete objects (or subobjects thereof), the < operator does not
       establish a strict weak ordering ([tab:cpp17.lessthancomparable],[expr.rel]) – end note*]

— *Bump existing remarks below new remark 7*

33.5.8 Class template atomic [atomics.types.generic]

33.5.8.3 Specializations for integers [atomics.types.int]

— *Add following public functions, immediately below* `fetch_xor:`

```
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 146, [tab:atomic.types.int.comp], add the following entries (note empty "Op" column):*

key	Op	Computation
max		maximum
min		minimum

— *Change remark 8:*

8   Remarks: ~~For~~ 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.

[*Note 2 : There are no undefined results arising from the computation. —- end note*]

— *Add remark 9 immediately below:*

9   Remarks: For `fetch_max` and `fetch_min`, the maximum and minimum computation is performed as if by `max` and `min` algorithms [alg.min.max], respectively, with the object value and the first parameter as the arguments.

— *Bump existing remarks below new remark 9*

33.5.8.5 Partial specialization for pointers [atomics.types.pointer]

— *Add following public functions, immediately below* `fetch_sub:`

```
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 147, [tab:atomic.types.pointer.comp], add the following entries (note empty "Op" column):*

key	Op	Computation
max		maximum
min		minimum

— *Add remark 9 with note 2 immediately below remark 8:*

9   *Remarks*: For `fetch_max` and `fetch_min`, the maximum and minimum computation is performed as if by `max` and `min` algorithms [alg.min.max], respectively, with the object value and the first parameter as the arguments.

     [*Note 2: If the pointers point to different complete objects (or subobjects thereof), the < operator does not establish a strict weak ordering ([tab:cpp17.lessthancomparable],[expr.rel]) – end note*]

— *Bump existing remarks below new remark 9*

14   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

[Github] Al Grant, Bronek Kozicki, and Tim Northover. Atomic maximum/minimum.
https://github.com/Bronek/wg21-p0493

[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

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

[N4917] Thomas Köppe. 2022-09-05. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4917

[P3008] Gonzalo Brito Gadeschi and David Sankel. P3008 : Atomic floating-point min/max.
https://wg21.link/p3008