Revising Atomic Max/Min Operations

Revising Atomic Max/Min Operations by Extending compare_exchange with Custom Comparisons & alternative solutions

Document number: P3454R0

Date: 2024-10-15

Project: WG21, SG1

Authors: Michael Wong, Gonzalo Brito, Fedor Pikus, Maged Michael

Reply to: Michael Wong <fraggamuffin@gmail.com>

Abstract

This paper proposes extending the `compare_exchange` functions in the C++ Standard Library to accept a custom comparison object, allowing for more flexible atomic operations. It supports a wide range of use cases, including conditional max/min operations, and offers a general solution to optimize atomic operations without introducing multiple new specialized atomic functions. Additionally, we present an alternative approach with conditional writes for `fetch_max` and `fetch_min`, comparing the benefits and trade-offs.

Revision History

R0 (2024/10/15: Wroclaw 2024 pre-mailing): Initial proposal

Introduction

Current atomic operations in C++ provide fixed operations (e.g., `fetch_add`, `fetch_or`) and the flexible but sometimes cumbersome `compare_exchange`. This proposal extends the flexibility of `compare_exchange` by allowing custom comparisons, which enables conditional

atomic operations without the need to introduce multiple new specific atomic functions like `fetch_min` or `fetch_max`.

P0493R5 proposes adding atomic max and min operations to the C++ standard library. While this addition is valuable, we believe the current design can be improved to better serve performance-critical concurrent code. This paper suggests modifications to the semantics and interface of these operations.

The paper's justifications for the current design, while valid, might not outweigh the potential advantages of our proposed change. Implementation flexibility, could be enhanced by allowing implementations to omit the store. The consistency argument, while understandable, could be addressed by considering alternative naming conventions that better reflect the operation's semantics.

Regarding portability, our observation about the potential for performance disparities across different architectures and Acknowledging these differences and allowing for optimized implementations could lead to more efficient and portable code in the long run.

The proposal's emphasis on performance considerations aligns with our perspective. Atomic operations are often used in performance-critical scenarios, and any unnecessary overhead can be detrimental. Let me summarize our arguments:

- Key Arguments:
    - Unnecessary writes (Gratuitous Store) are performance bottlenecks
    - Existing synchronization patterns cover edge cases
    - Implementation flexibility favors conditional writes
    - Consistency with other fetch operations may be misleading
- Proposed Semantics:
    - Allow Conditional write: allow only updating the memory location if the comparison condition is met (e.g. new value is larger for max)
    - OR Extend `compare_exchange_strong` and `compare_exchange_weak` with a new overload accepting a Comparison object
    - Dual memory ordering: Separate for success and failure paths
- Implementation Flexibility:
    - Allowing Conditional writes enablesfor more efficient implementations
    - Hardware-specific optimizations become possible
- Synchronization Considerations:
    - Release semantics still maintained when necessary
    - Consistent with typical producer-consumer patterns
- Naming Considerations:
    - "fetch_max" may not accurately reflect conditional nature
    - Alternative: “conditional_max” "compare_select_max" or similar
    - Extend `compare_exchange_strong` and `compare_exchange_weak` with a new overload accepting a Comparison object
- Proposed new fetch_max semantics

Aspect	Current Proposal	Suggested Revision
Write Behavior	Always write	Conditional write
Memory Ordering	Single parameter	Dual parameters
Performance Focus	Moderate	High
Implementation Flexibility	Limited	Increased

Motivation and Scope

The primary motivations for these changes are:

1. Performance optimization by avoiding unnecessary writes
2. Increased implementation flexibility across different architectures
3. Better alignment with real-world usage patterns of atomic operations
4. More precise control over memory ordering for advanced users

Unnecessary Write (Gratuitous Store) Issue:

- The argument we present is that the unconditional store in `fetch_max`, even when no update is necessary, is artificial and introduces unnecessary costs. The conditional nature of `fetch_max` should allow for cases where no store occurs if the current maximum is already greater or equal to the new value.
- From a program correctness standpoint, you explain that the program's memory consistency model (i.e., the use of release barriers on prior stores) guarantees that the data is already in a valid state, so there is no need for an additional write if the maximum hasn't changed.

Alignment with Existing Producer-Consumer Patterns:

- In producer-consumer models, the releasing of data (such as setting a max value) is already done by other operations like `fetch_add` or `fetch_max`. Once the maximum value is set by a previous operation, an additional release when no value has changed is

redundant, and the previous operation has already ensured the data's visibility to consumers.

Implementation Flexibility:

- Our suggestion allows implementations to either perform a conditional store or not, depending on the hardware. For architectures where a conditional store could yield better performance, like ARM, this flexibility would allow for optimizations.
- By contrast, the current proposal forces implementations into an unconditional write path, limiting the flexibility to optimize for different architectures.

Naming Semantics:

- We argue that part of the issue stems from the naming. The name `fetch_max` implies certain expectations, but the behavior might be better served by an alternative name that reflects its actual conditional nature, such as conditional_max or `compare_select_max`. This would better match what the operation does: selecting the maximum, updating it conditionally, and returning the maximum, regardless of whether the value changed or not.
- The name choice seems to be a trade-off between consistency with other `fetch_XXX` operations and more precise behavior.
- Extend `compare_exchange_strong` and `compare_exchange_weak` with a new overload accepting a Comparison object

Portability vs. Performance:

- On ARM, where atomic max/min instructions are available, this proposal gives developers direct access to that functionality. But on x86, where a release-read barrier is more expensive, developers who prioritize performance would likely avoid using the standard `fetch_max` in favor of writing a custom implementation that performs better.
- The intent behind making `fetch_max` consistent across architectures may actually hurt its portability, as developers on x86 may avoid it due to performance reasons.

Performance-Centric Design:

- We argue that the primary reason for using atomics is performance, and if an operation like `fetch_max` sacrifices performance for consistency or simplicity, it defeats the purpose of using it in the first place. Atomics are, by nature, complex and non-intuitive, and their usage is already difficult, so performance should take precedence over consistency or naming concerns.
- For example, in concurrent algorithms for shared data structures or lock-free queues, where values are frequently compared but less frequently updated, forcing unnecessary writes introduces overhead that harms scalability.

Motivation and Scope

The primary motivations for this change are:

1. Enable more flexible atomic operations without proliferating specific atomic functions
2. Support conditional update patterns (like max/min) efficiently
3. Improve forward progress in concurrent algorithms by reducing the need for compare_exchange loops
4. Provide a general mechanism that can adapt to various use cases

Proposed Changes

1. Extend `compare_exchange_strong` and `compare_exchange_weak` with a new overload accepting a Comparison object:

```
template<class T, class Comparison>
bool compare_exchange_strong(Comparison&& cmp, T& expected, T desired,
                             memory_order success, memory_order failure) noexcept;
```

2. Modify the specification of compare_exchange to use the provided comparison: It then atomically evaluates `cmp(*ptr, expected)`, and if true, [...]
3. Specify the behavior for existing overloads: For the overloads without cmp, the comparison returns true if the value representation of its arguments is the same and false otherwise.
4. This change avoids redundancy by providing a generalized mechanism for comparison-based atomic updates, which can be adapted to various use cases such as max/min comparisons, complex comparison-based operations, and custom synchronization logic.

Examples

```
std::atomic<int> atom{5};
int expected = 3;
int desired = 10;

// If atom is strictly less than expected, update to desired
atom.compare_exchange_strong(std::less{}, expected, desired,
                             std::memory_order_acq_rel,
std::memory_order_acquire);

// Implementing fetch_max
auto fetch_max = [&atom](int val) {
```

```
    int expected = atom.load();
    while (!atom.compare_exchange_weak(std::less{}, expected, val)) {}
    return expected;
};
```

Design Considerations

Comparison Object Requirements

The Comparison object should be a callable object accepting two arguments of type `T const&` and returning a `bool`. It must not modify its arguments or have side effects (stateless). This design keeps the atomic library simple, extensible, and in line with existing functional paradigms in C++

Performance

This approach avoids the need for compare_exchange loops in many scenarios, potentially improving forward progress in concurrent algorithms. Hardware-specific optimizations can still be applied for common comparison operations.

Backward Compatibility

Existing compare_exchange overloads remain unchanged, ensuring full backward compatibility.

Alternative Solution: Conditional fetch_max and fetch_min

An alternative approach to solving the original problem of efficient atomic max/min operations was also considered. This section outlines this alternative and compares it with the main proposal.

This alternative solution could coexist with the custom comparison approach for scenarios where direct hardware-supported atomic operations (e.g., atomic max/min) are essential, while the custom comparison approach addresses broader, more complex atomic patterns.

Alternative Proposed Changes

1. Modify fetch_max and fetch_min to perform conditional writes:
    - Only update the atomic variable if the new value would change the result
2. Introduce dual memory ordering parameters:

    - Similar to compare_exchange, add separate memory orders for success and failure cases
3. Consider renaming the operations to better reflect their semantics:
    - Potential names: compare_select_max, conditional_max

Example of Alternative Approach

```
T conditional_max(T value,
        memory_order success = memory_order_seq_cst,
        memory_order failure = memory_order_seq_cst) volatile noexcept;
```

Comparison of Approaches

Aspect	**Custom Comparison Approach compare_exchange**	**Conditional fetch_max/min Approach**
Generality/Flexibility	Highly general, supports various comparison operations Supports a wide range of custom comparisons beyond min/max (e.g., `greater_than`, `less_than`, complex logic)	Specific to max/min operations
Complexity	Introduces new concept (Comparison object)	Relatively simple extension of existing operations
Performance	Potential for optimization, but may vary. The custom comparison solution provides better scalability in more complex multi-threaded scenarios or in environments where more flexibility is needed Avoids unnecessary writes, reducing contention and improving performance	Direct mapping to hardware instructions on some platforms. The conditional `fetch_max`/`fetch_min` approach could have the advantage in environments where hardware support for atomic max/min is readily available (e.g., ARM) Also avoids unnecessary writes but is more focused on min/max comparisons

Consistency/API Simplicity	Consistent with existing compare_exchange semantics Reuses existing `compare_exchange` API with an additional `Comparison` parameter	Introduces new semantics for specific operations Requires changes to the `fetch_max`/`fetch_min` API, but avoids adding new functions
Extensibility	Easily extensible to new operations. Supports a wide range of use cases beyond atomic min/max	Limited to max/min, may require future proposals for other operations Specifically designed for min/max operations
Implementation	May require more complex implementation	Potentially simpler implementation, especially on supporting hardware
Usability	Requires understanding of Comparison objects Requires developers to understand and use custom comparison objects, potentially increasing complexity	Straightforward for users familiar with existing fetch operations More intuitive for those familiar with `fetch_max`/`fetch_min`, especially with potential renaming
Memory Ordering	Supports dual memory ordering (success/failure) through `compare_exchange`	Adds dual memory ordering to `fetch_max` and `fetch_min`
Backward Compatibility	Fully backward compatible with existing `compare_exchange` behavior	Minor API changes but backward compatibility is maintained
Naming and Semantics	No renaming necessary, but requires understanding of custom comparisons	Renaming (`compare_select_max`, `conditional_max`) can make the purpose clearer

Advantages of Custom Comparison Approach

1. Provides a general solution that can handle a wide range of atomic update patterns beyond just max/min.
2. Maintains consistency with existing compare_exchange semantics.

3. Avoids proliferation of specific atomic functions in the standard library.
4. Allows for user-defined comparison operations, increasing flexibility.

Advantages of Conditional fetch_max/min Approach

1. Directly addresses the specific use case of atomic max/min operations.
2. Potentially more efficient on hardware with native atomic max/min instructions.
3. Simpler to use for the specific case of max/min operations.
4. Clearer semantics for the specific operations it covers.

Disadvantages of Custom Comparison Approach

1. Introduces a new concept (Comparison object) that users need to understand.
2. May be more complex to implement efficiently across all platforms.
3. Potential for misuse with incorrect comparison implementations.

Disadvantages of Conditional fetch_max/min Approach

1. Limited to max/min operations, doesn't solve the general problem of conditional atomic updates.
2. Introduces new semantics specific to these operations, potentially complicating the overall atomic API.
3. May lead to requests for similar conditional versions of other atomic operations in the future.

Conclusion

While both approaches have merit, this proposal recommends the Custom Comparison approach due to its greater generality and consistency with existing C++ atomic operation design. However, we acknowledge that the Conditional fetch_max/min approach may have advantages in specific use cases and on certain hardware platforms.

Both approaches could coexist in the standard, offering developers the choice between hardware-optimized conditional writes for common cases like max/min and the more general, flexible solution of custom comparison-based atomic operations.

We invite feedback from the committee on both approaches to guide further refinement of this proposal.

References

- - P0493R5: Atomic minimum/maximum

Acknowledgements

Thanks to Fedor Pikus for the initial idea. Also thanks to Gonzalo Brito and Maged Michael's analysis that led to this proposal.