Document number: P1382R0
Date: 20190111 (pre-Kona)
Project: Programming Language C++, WG21, SG1
Authors: JF Bastien, Paul McKenney, and the indefatigable TBD
Email: jfbastien@apple.com, paulmck@linux.ibm.com
Reply to: paulmck@linux.ibm.com

volatile_load<T> and volatile_store<T>

1. Introduction

This paper is an offshoot of P1152: Deprecating `volatile`, addressing the suggestion in Section 4 of that paper. It also draws upon Linux-kernel experience with `READ_ONCE()` and `WRITE_ONCE()`, as documented in P0124R6: Linux-Kernel Memory Model and in Section 4.3.4 ("Accessing Shared Variables") of Is Parallel Programming Hard, And, If So, What Can You Do About It?. This paper also owes its genesis to the realization of one of the authors (Paul) that a surprisingly large number of C++ Standards Committee members did not realize that correct operation of their credit cards depended on C and C++ compilers doing the right thing with `volatile`, as opposed to said compilers mindlessly complying with the relevant standards.

2. History/Changes from Previous Release

2018-12-16 [D1382R0] pre-Kona meeting

- Started.

---

3. Guidance to Editor

[This section will be filled out once wording has been debated to the point of rough consensus.]

4. Design Rationale, Goals, and Constraints

The `volatile` keyword has served the computing field well (if rather controversially) for many decades. So why change?

One reason was called out in the introduction: This keyword has done its job in spite of the standard. It would clearly be a great improvement if for the standard were to help rather than hinder dissemination of `volatile` knowledge.

Another reason is changes in hardware design of input/output (I/O) devices. The `volatile` keyword was conceived in a time when I/O devices were controlled either by special machine instructions or memory-mapped I/O (MMIO) locations. Use of special machine instructions to control I/O devices is declining, in part due to portability concerns. Such instructions were and should remain outside of the scope of the standard, but MMIO locations are and will likely remain a key motivator for `volatile`. What has changed is the partitioning of computing systems into different clock domains, with the core CPU running at much faster speeds than peripherals, and the consequent use of multiple levels of cache memory to accommodate this different in speed. This means that MMIO accesses, which must interact directly with peripheral devices, are quite slow, in particular, much slower than cache-mediated access to main memory.

Many modern device therefore use MMIO sparingly. One common approach is to make the I/O device participate in the main-memory cache-coherence protocol, and then to place arrays of *control blocks* (CBs) in main memory. Each element of such an array describes one I/O request, including the address and length of the block of memory to be output from or input to, along with any required control information (for example, a bit indicating that the array element is ready for device processing, another bit indicating that the corresponding I/O has completed, and a third bit indicating the last element of the array). A single MMIO operation informs the device of the location of this array, and the overhead of this single operation is then amortized over all the I/O operations specified by the full array, reducing per-I/O MMIO overhead to a value arbitrarily close to zero.

Device drivers need not use the `volatile` keyword when initializing a new array of CBs because the device is not yet aware of its existence. This allows both the compiler and the cache/store-buffer hardware to fully optimize these initialization accesses. Once initialization is complete, these accesses must be finalized. Such finalization is not always portable, but might be as straightforward as a full memory-fence instruction. Then the MMIO instruction informs the device of the newly initialized array, and causing that device to commence I/O operations. However, when polling array entries for completion, the driver absolutely must use `volatile` reads because the compiler is unaware of the device's memory writes. Therefore, use of the `volatile` keyword on the array object is not consistent with maximal performance. This provides motivation for the `volatile_load<T>` facility described in this document.

This same scenario also provides motivation for `volatile_store<T>`. To see this, consider the C++ code that implements the device firmware in the above example.

Given that shared-memory communication with an I/O device motivates both `volatile_load<T>` and `volatile_store<T>`, it should not be too great a leap to see how they can be used for shared-memory concurrent programming, as `READ_ONCE()` and `WRITE_ONCE()` in fact are used within the Linux kernel. Many might prefer use of C++11 atomics, but experiences thus far have been mixed, optimizations can prevent their use in low-level code, some recent proposals for changes to relaxed atomics might be even less consistent with such use, and much concurrent code written before the advent of C++11 atomics will continue to use `volatile` for some time to come. In addition, initialization/cleanup scenarios similar to that described above for the device driver also exist in concurrent code, which motivates some way of providing lower-overhead access for single-threaded access during initialization and cleanup operations.

What must `volatile_load<T>` and `volatile_store<T>` do?

`volatile_load<T>` might produce any object representation of the underlying type, as expected given that some code unknown to the compiler might be storing to the object in question. In particular, that unknown-to-the-compiler code might have repurposed padding, as often happens with later revisions of I/O devices. Additionally, `volatile_load<T>` can result in side effects, again as expected given that some code unknown to the compiler might be activated as a result of a hardware response to an MMIO load. Finally, the semantics of `volatile_load<T>` depend on access size, meaning that tearing must be avoided where possible given the available "pure" load instructions. For its part, `volatile_store<T>` can result in side effects over and above that of the store itself, again as expected given that some code unknown to the compiler might be polling the stored-to object. And as with `volatile_load<T>`, the semantics of `volatile_store<T>` depend on access size, again meaning that tearing must be avoided where possible given the available "pure" store instructions. Both `volatile_load<T>` and `volatile_store<T>` may be used to control order of accesses with respect to signals and special C functions that affect control flow.

5. Proposed wording

?.1 Volatile Accessor [vol-acc]

1. The `volatile_load<T>` and `volatile_store<T>` templated free functions allow specific accesses to given object to be carried out with volatile semantics:
    a. Implementations must not tear accesses for which load (`volatile_load<T>`) or store (`volatile_store<T>`) instructions for that access's size and alignment are available. [Note: This implies that use of both `volatile_load<T>` and `volatile_store<T>` are only semi-portable, which is entirely consistent with the

observed fact that device drivers are only semi-portable. It also implies that implementations are not obliged to manufacture load or store instructions, for example, from atomic compare-and-swap instructions. -- End note.]

b. A `volatile_load<T>` invocation may produce any object representation from the underlying type. [Note: The implementation cannot assume that global analysis can result in valid constraints of the objects produced, including the values of any padding. The `volatile_load<T>` is after all telling the implementation that it lacks the information required to derive such constraints. -- End note.]
c. Both `volatile_load<T>` and `volatile_store<T>` can produce side effects.
d. Use of `volatile_load<T> or volatile_store<T>` on unaligned objects is implementation-defined.
e. Use of `volatile_load<T> or volatile_store<T>` on objects whose size does not correspond to a load instruction (`volatile_load<T>`) or store instruction (`volatile_store<T>`) is implementation-defined.
f. Code for `volatile_load<T>` and `volatile_store<T>` is emitted in program order. [Note: Implementations are not obliged to provide cross-thread ordering for instances of `volatile_load<T>` and `volatile_store<T>` in the absence of other mechanisms. One such mechanism is `atomic_thread_fence`. -- End note.]
g. Implementations must not elide or fuse instances of `volatile_load<T>` and `volatile_store<T>`.
h. Implementations must not speculate or insert instances of `volatile_load<T>` or `volatile_store<T>` into a program. [Note: The as-if rule does not apply here because MMIO accesses have implementation-defined semantics. -- End note.]
i. A `volatile_store<T>` access does not constitute a data race with either a `volatile_load<T>` or a `volatile_store<T>` if both occur in the same thread, even if either or both of them occur in a signal handler.
j. Both `volatile_load<T>` and `volatile_store<T>` may be used to prevent code motion around special library functions such as `setjmp` and `longjmp` [csetjmp.syn].

2. An object that is accessed with either `volatile_load<T>` or `volatile_store<T>` may also be accessed using normal C++ loads and stores. Code emitted for any normal load or store must follow that for any program-order-previous `volatile_load<T> or volatile_store<T>` to that same object. Similarly, code emitted for any normal load or store must precede any program-order-later `volatile_load<T> or volatile_store<T>` to that same object.

Header <vol_access> synopsis

```
namespace std {
namespace experimental {
```

```
// ?.1.1 function template volatile_load
template<typename T>
T volatile_load(const T* p);

// ?.1.2 function template volatile_store
template<typename T>
void volatile_store(T* p, T v);
} // namespace experimental
} // namespace std
```

?.1.1, function template volatile_load [vol-acc.load]

```
template<typename T>
constexpr T volatile_load(const T* p);
```

1. Mandates: `T` shall satisfy the requirements of `TrivallyCopyable`.
2. Effects: Causes the value of `*p` to be returned.

?.1.2, function template volatile_store [vol-acc.store]

```
template<typename T>
constexpr void volatile_store(T* p, T v);
```

3. Mandates: `T` shall satisfy the requirements of `TrivallyCopyable`.
4. Effects: Causes the value of `*p` to be overwritten with `v`.

6. Acknowledgements

The authors thank TBD people for TBD.

7. References

[P0098R0] Towards Implementation and Use of memory_order_consume

[P0124R6] Linux-Kernel Memory Model

[P1152R0] Deprecating volatile

[P1217R0] Out-of-thin-air, revisited, again

Time to move to C11 atomics? Jonathan Corbet, Linux Weekly News.

Is Parallel Programming Hard, And, If So, What Can You Do About It? Paul E. McKenney, editor.