N1909
Note 1 Clarification for **atomic_compare_exchange**

Blaine Garst
2014-11-11

In **7.17.7.4 The atomic_compare_exchange generic functions** paragraph 3 states

NOTE 1 For example, the effect of **atomic_compare_exchange_strong** is

```
    if (memcmp(object, expected, sizeof (*object) == 0)
         memcpy(object, &desired, sizeof (*object));
    else
         memcpy(expected, object, sizeof (*object));
```

The goal for this note was to show that either object or expected was updated rather than just being a conditional operation on object alone. It is being read by some parties, however, to mean that atomic_compare_and_exchange is intended to do bit comparison instead of value comparison. This is an erroneous reading.

Consider first non-lock-free atomic types. These obviously require use of the lock, whether inline or in an external table. So the first conclusion is that an implementation must already select different implementations for these generic functions based on whether the type is lock-free or not (ignoring lock bits leads to data races). The basic algorithm is to take the lock on the target object, extract and compare values with expected, and store or update desired as appropriate, and release the lock. The extraction and comparison would likely be done by the compiler through the use of type specific intrinsics that may or may not get inlined by the optimizer.

Consider second the cases of padded integer types, padded struct or union types, and float types.. All of these types have multiple bit representations for one or more values and will fail erroneously when object and expected differ in representation but not value. An implementation should, as for non-lock-free data types, select an appropriate intrinsic to perform this operation. There are two basic choices for the intrinsic. First, make all these atomic types locking, and use the locking strategy already in place to attain the lock and extract and compare the value bits appropriately. An alternate strategy might be to first "normalize" `*object` and `*expected`, and then perform bitwise compare and exchange.

To support this conclusion, I propose clarifying the note to apply to unpadded lock-free integer types.

Proposed Technical Corrigendum

In **7.17.7.4 The atomic_compare_exchange generic functions** paragraph 3 replace

> NOTE 1 For example, the effect of **atomic_compare_exchange_strong** is

with

> NOTE 1 For example, the effect of **atomic_compare_exchange_strong** is, for unpadded lock-free integer types, atomically