M A Y 2 5 , 2 0 1 0

Draft 7

ATOMIC PROPOSAL, N1476

BLAINE GARST

APPLE INC.

blaine@apple.com

Introduction

The C language community has for decades explored concurrency in the form of threads, mutexes and condition variables to describe all grain sizes of parallelism, and these are now reflected in the proposed N1425 C1x Standard. Also reflected is a new formalism for describing the memory model and in particular a new atomics library for describing operations on designated atomic memory objects.

The vast majority of modern multi-cpu architectures provide atomic compare-and-swap, test-and-set, or other forms of atomic operations on memory. These individual instructions coordinate with the caches to do correct behavior to the backing memory. Across processors, however, memory read and memory store barriers must be used to see these results reliably.

The atomics library of section 7.16 is intended to be an exact subset of the proposed C++0x N3092 Final Committee Draft section 29. What is missing, however, is direct language treatment of sequentially-consistent atomic behavior for C language objects, as does occur in C++0x via templates and operator overloading. As such C1x programmers are forced to exclusively use the library directly and its limited set of types. Of particular concern, for example, is the lack of ability to define atomic structures.

The proposal is to add an _Atomic type qualifier and to have it have the following effects:

1. all compound assignment operators (e.g. +=, /=, ^= , ...,) and all forms of ++ and -- are overloaded to provide sequentially-consistent atomic behavior, whether achieved directly via hardware instructions , by small inline compare-and-swap code sequences, or by support from compiler intrinsics, as the implementation decides. Thus, even atomic float identifiers can make use of sequentially-consistent operators..

2. Ordinary read and write access to _Atomic identifiers is sequentially-consistent, requiring memory-load barriers as necessary on the hardware before every read, and memory-store barriers after writes so that dependent data can be properly provided.

3. A new ?= : ternary assignment operator is introduced to express an assign value if identifier has the specific value operation. When used upon an atomic qualified identifier, a sequentially-consistent compare-and-swap operation will be performed. This primitive can be used, for example, to implement a simple spin-lock that guards changes to other objects.

_Atomic applies uniformly to scalars, arrays, and aggregates, yet has little to no meaning for function results and parameters. An atomic structure would be read and written atomically using intrinsic helper functions, members not marked _Atomic use unchanged access rules.

Modifications

The following are the more formal descriptions of changes necessary to the **N1425** proposed draft standard to reflect the preceding high level descriptions of atomic.

Section 5.1.2.4

para5: The _Atomic type qualifier designates objects as being atomic and as requiring sequentially-consistent operations directly on these objects. Additionally, the library....

1. Section 6.2.5 para 26

“and **restrict** qualifiers.” Further, there is the _Atomic qualifier which may combine with **volatile** and **restrict**. [Note: A locking implementation of atomics may be required on architectures where non-natural alignment precludes use of native atomic instructions]

2. Section 6.2.6.1

When a value is stored into an atomic object it must be done such that the store is done via sequentially-consistent semantics, e.g. a memory-store-barrier is required. When an atomic object is read it must also be done via a sequentially-consistent operation, e.g. a memory-load-barrier must be issues as necessary on the hardware architecture.

3. Section 6.4.1,

Add _Atomic to the list of keywords.

4. Section 6.4.6

Add ?= to list of punctuators.

5. Section 6.5.2.4

Postfix ++ and -- operations on atomic objects are atomic read-modify with sequentially-consistent memory order. [Note: where pointers to atomic objects can be formed, these operations are equivalent, where T is the type of E1, to the code sequence

```
T result;
T expressionValue = E1;
do
    result = expected OP 1;
while (!atomic_compare_exchange_strong(&E1, & expressionValue, result);
```

where *expressionValue* is the result of the operation.]

6. Section 6.5.3.1

Prefix ++ and -- operations on atomic objects are atomic read-modify with sequentially-consistent memory order. [Note: where pointers to atomic objects can be formed, these operations are equivalent, where T is the type of E1, to the code sequence

```
T result;
T expected = E1;
do
    result = expected OP 1;
while (!atomic_compare_exchange_strong(&E1, &expected, result);
```

where *result* is the result of the operation.]

7. Section 6.5.16.2 Compound Assignment, para 3, Semantics

If E1 is atomic, the operations are atomic read-modify with sequentially-consistent memory order. [Note: where pointers to atomic objects can be formed, these operations are equivalent, where T is the type of E1, to the code sequence

```
T result;
T expected = E1;
do
    result = expected OP E2;
while (!atomic_compare_exchange_strong(&E1, &expected, result);
```

where *result* is the result of the operation.]

8. Section 6.5.16.3 The ternary assignment operator ?= :

The ternary operator *lvalue ?= candidateExpr : replacementExpr* tests that the *lvalue* has *candidateExpr* value and if so assigns the *replacementExpr* value, yielding a boolean value affirming or denying that the assignment occurred. If lvalue is atomic, the test and assignment must be performed with sequentially-consistent behavior, e.g. the test-and-set must be done atomically, with memory-load and memory-store barriers.

[Example: lazy initialization

```
_Atomic struct x *GlobalPX = NULL;
```

```
struct x *getGlobalPX() {
    struct x *returnValue;
    // the read of GlobalPX synchronizes with assignment to member
    if (! (returnValue = GlobalPX)) {
          struct  x *tmp = (struct x *)malloc(sizeof(struct x));
          tmp->member = value;
          if (GlobalPX ?= NULL : tmp)
              returnValue = tmp;
          else
              free(tmp);
    }
    return returnValue;
}
```

9. Section 6.7.3 Type Qualifiers

Paragraph 1: Add _Atomic to the list.

Paragraph 2: (Constraints)

There shall not be atomic members (either direct or recursively) of unions.

Paragraph 5 If an attempt is made to refer to an object defined with an _Atomic-qualified type through use of an lvalue with non-_Atomic-qualified type, the behavior is undefined.

(new paragraph) The atomics library functions of 7.16.6 apply to pointers to _Atomic-qualified types as they do to pointers to the atomic integral and pointer types defined in 7.16.5.

(new paragraph) The _Atomic qualifier when applied to a structure implies such qualification to each member. [Note: A *data-race* may occur if access to the entire structure in one thread conflicts with access to a member from another thread, where one or more of such accesses is a modification, and such a *data-race* results in undefined behavior].

10 7.16.3 para 2, 3;

spelling of "heed" is wrong.