Document number: P2061R0
Date: 2020-01-12 (pre-Prague)
Reply to: David Goldblatt <davidtgoldblatt@gmail.com>
Audience: SG1

Sequential consistency for atomic memcpy

Background

In Belfast, there was some discussion as to whether or not the atomic memcpy functions suggested in P1478 should allow `memory_order_seq_cst` as their memory ordering argument (i.e. with the store to or load from each byte appearing in the total order of all SC operations). I think SG1 was eventually mostly on board with allowing it, but only so long as it didn't require a fence per load/store. I'll recap the argument for allowing them and argue that they can be efficiently implemented. The trick will be to regard the loads and stores within an atomic memcpy call unsequenced with respect to one another (like if executed with an `std::execution::unseq` policy).

Why bother?

The most common use case we anticipate for atomic memcpy functions is in seqlock implementations, which only needs acquire or relaxed ordering. So why bother allowing something stronger?

The algorithmic argument

No algorithm *needs* seq_cst semantics for the atomic_memcpy, in the sense that you can always use fences to get the same effect. Then again, that's true for acquire semantics, too. (In fact, you don't need atomic_memcpy at all; you can get the same effect and performance profile with an array of word-sized atomics accessed with relaxed semantics). There are some places where it's the most natural way of expressing a synchronization pattern:

- A seqlock-like abstraction that allows writer-writer contention: each thread stores into `some_boolean_array[tid]` to indicate it accessed the protected contents. It reads the entire array to see if any other thread (and presumably goes down some contention-management slow path if so -- you can imagine something like an STM implementation where stores proceed to the underlying data optimistically, and detected conflicts are resolved at commit time).

- A counting thread maintains a counter and a threshold, and goes down a slow path when the first exceeds the second (and resets the threshold to something higher). A remote thread can send the counting thread down the slow path by zeroing the threshold. The modifications of the threshold need to be seq_cst (to ensure that, if in a race the counting thread overwrites a remote-thread reset, it will still whatever state the remote thread set before its modification). You could do this with atomics, but shouldn't if the threshold is 64 bits and your machine doesn't have lock-free 64 bit atomics.

The moral one

The memory model is simpler and more comprehensible when the ordering applied to an operation is decoupled from the type of operation as much as is possible. Each special case is a little bit more ugliness in the world.

Correctness of a cheap implementation

The argument against allowing seq_cst atomic memcpys was that it seemed it would require a fence per word-sized load or store. Here, we'll see that a user program can't observe the difference between two implementations:

- N unsequenced seq_cst loads or stores
- A seq_cst fence, N relaxed loads or stores, and another seq_cst fence

And so we can as-if-rule the first into the second. We thereby bound the cost to at most two fences total. Once we're talking about real hardware, of course, it's possible to do better (the obvious mfence-only-after-the-stores strategy is correct on x86, for instance). The reason for the two-fence approach is that it lets us frame the argument in terms of the rest of the memory model, rather than in terms of hardware.

Consider an execution with the two-fence implementation strategy. Label the operations in its SC ordering as going from 1 to N. Say that `begin(i)` is true if operation i is an SC fence at the beginning an atomic memcpy, and `end(i)` is true if operation i is an SC fence at the end of one. If `begin(i)` or `end(i)` if true, let `operations(i)` be the associated (relaxed) loads or stores. Given a load or store A in an atomic memcpy, let `lead(A)` be the SC fence sequenced before it, and `trail(A)` be the SC fence sequenced after it. Run the following algorithm to emit a new ordering, SC'.

```
Initialize SC' to an empty list
Initialize pending_ops to an empty set
for i in 1 ... N:
  if begin(i):
    add operations(i) to pending_ops
  else if end(i):
    for each x in the intersection of operations(i) and pending_ops:
      for y each coherence predecessor of x (in order) in pending_ops:
```

```
      add y to SC’ and remove it from pending_ops
    add x to SC’ and remove it from pending_ops
else:
  Add i's operation to SC’
```

I claim that the execution obtained by replacing the SC ordering with SC’ is a valid one for the all-seq_cst-loads/stores implementation. Obviously it contains all the same operations; we just need to show we didn’t introduce any rule breakage. There are only two tricky cases:

SC’ includes HB restricted to the seq_cst operations: for the operations in SC this is trivial. The others must be in atomic memcpy calls. Take A and B in such calls. Note that if $A <_{HB} B$, then we must have $A <_{SB} \text{trail}(A) <_{HB} \text{lead}(B) <_{SB} B$ (or at least, can assume so because of the as-if rule -- the user can’t prove any stronger HB relationships without making assumptions about implementation internals). So A is emitted into SC’ before B.

SC’ includes CO restricted to the seq_cst operations: Take A and B on the same object, with $A <_{CO} B$. If A and B are not one of the operations in an atomic memcpy, then they’re in SC and we get its guarantee. If not, consider when B was emitted into SC’; at the time we process its trailing fence. We must have previously processed A’s preceding fence ($\text{lead}(A) <_{SB} A <_{CO} B <_{SB} \text{trail}(B)$, and so $\text{lead}(A) <_{SC} \text{trail}(B)$), and so A was added to `pending_ops`. A is therefore added to SC’ before B, and so $A <_{SC'} B$.

This argument isn’t terribly formal, and glossed over some details:

- On real implementations, we may be dealing with mixed-size accesses, which the rest of the memory model doesn’t address at all.
- Our claim that all operations in SC’ are either in SC or part of an atomic memcpy wasn’t actually true; they could have been part of an atomic_ref operation. But the atomic_ref exclusivity requirements mean that there has to be an HB relationship between atomic_ref accesses and others, so they don’t “really” participate.

That being said, I think it’s passed my threshold for “likely correct, especially if we’ll discover mistakes through the TS process”.