SCFence: Automatic Inference of Memory Order Parameters to Obtain SC Behaviors under C/C++11

Peizhao Ou and Brian Demsky

University of California, Irvine

{peizhaoo,bdemsky}@uci.edu

Abstract

1. Introduction

2. Motivating Example

3. Technical

3.1 Inference Rules

Previous work takes advantage of model-checking approach to check whether a specific C/C++11 trace is sequentially consistent. By building up edges (*sb*, *rf* and *sc* by implication rules) between atomic operations, it judges whether the trace is SC by whether there exists a cycle in that graph. Besides, when it finds a non-SC trace, it has a sorting algorithm that generates an SC-like trace and exposes which reads-from edge that causes a cycle.

Under the C/C++11 memory model, inferring the ordering parameters to obtain SC behaviors is essentially a searching problem. In the absence of consume operations, memory order parameters for atomic operations can be only one of the following: *memory_order_relaxed*, *memory_order_release*, *memory_order_acquire*, *memory_order_acq_rel* and *memory_order_seq_cst*. By enumerating all possible memory order parameters, we can guarantee that we can find out all the possible inference of parameters that ensure SC behaviors for a specific test case. However, this naive approach obviously leads to an exponential searching space.

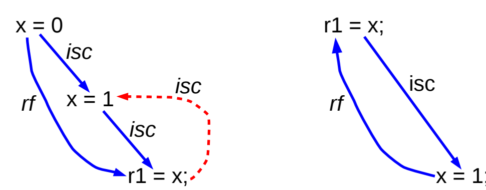
Actually, when we have a non-SC execution, we have some knowledge availabe reflecting where the problem may lie. Consider we start from the case where all memory order parameters are *memory_order_relaxed*. Whenever the model-checking approach finds out a cycle in a specific execution, we have to infer some stronger memory orders to eliminate the cycle. What causes the cycle to happen leads to the non-SC trace. We propose a search-based approach combined with cycle patterns and their fixes to reduce searching space.

Among those non-SC executions, previous work will reorder the original trace to yield an SC-like trace which indicates where a bad read-from edge happens. In Figure 1 , we show the two universal patterns that can exist in cycles. We explain how we should fix the cycle patterns respectivelys the following.

Old value read: *Patter a* in Figure 1 shows the case where a load reads its value from an old store rather than the recent store. *isc* is the union of *sc*, *hb*, *rf* and the implied edges. If any one of the two *isc* edge is not *hb*, we will impose either *hb* or *sc* to it. If an *isc* edge is the union of *rf*, *hb*, we can use release/acquire pair to establish synchronization; otherwise, we will make both operation sequentially consistent. For the latter case, it can be insufficient to get rid of the execution. However, by proritizing the *sc* edges in building the graph, a different problematic spot will be found, and we can fix the cycles iteratively. Also, when both *isc* edges are implied edges, we need to explore the cases where there exist other loads that are *isc-ordered* after the old store and before the read.

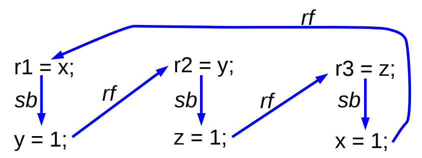
Future value read: *Pattern b* in Figure 1 shows a load that is *isc-ordered* before a store reads its value from that store (or reading the future value). The two possbile fixes is: 1) impose *hb* between the store and the load, which can be easily done by using release/acquire; 2)impose *hb* or *sc* between the load and the store, which is similar to what we discussed in *pattern a*.

Figure 3 shows the core searching algorithm for all possible parameters.

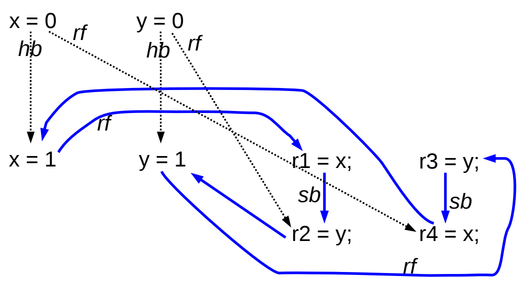


Old Value Read *(pattern a)*　　Future Value Read *(pattern b)*

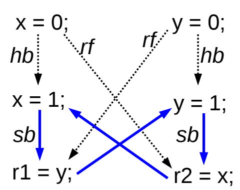
Figure 1. Cycle Patterns for Non-SC Behaviors



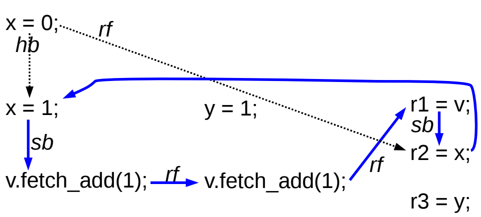
Circular Reads-from



Independent Reads & Independent Writes



Peterson Lock



Release Sequence

Figure 2. Non-SC Examples

4. Evaluation

5. Related Work

SC [1]

```
function INFERPARAMS
    candidates := {}
    candidate c1 := replace all wildcards with relaxed
    candidates += c1
    results := {}
    while candidates is not empty do
        Candidate c := candidates.pop()
        Model-check with c and yield a cycle l
        if l == NULL then
            results += c
        else
            STRENGTHENPARAM(l, c, candidates)
        end if
    end while
    return results
end function
procedure STRENGTHENPARAM(c, p, candidates)
    while ∃ a pattern p in cycle c do
        possible_fixes := strengthen c by pattern p
        candidates += possible_fixes
    end while
end procedure
```

Figure 3. Algorithm for Searching All Possible Parameters

6. Conclusion

References

[1] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979.