

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 available 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 2, we show the two universal patterns that can exist in cycles. We

explain how we should fix the cycle patterns respectively the following.

Old value read: *Pattern a* in Figure 2 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 prioritizing 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 2 shows a load that is *isc-ordered* before a store reads its value from that store (or reading the future value). The two possible 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.

4. Evaluation

5. Related Work

SC [1]

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.

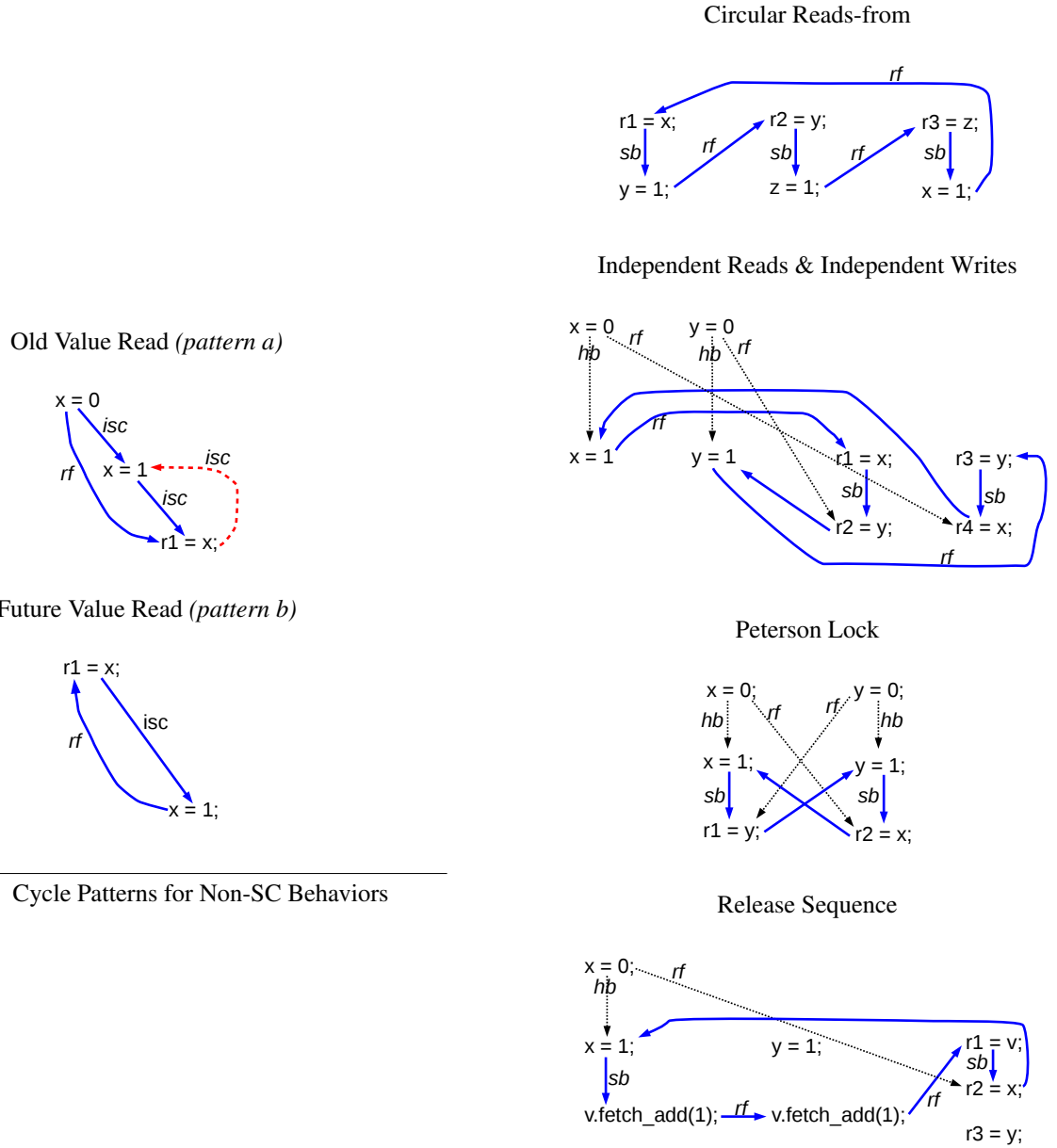


Figure 1. Cycle Patterns for Non-SC Behaviors

Figure 2. Non-SC Examples

```

1: function INFERPARAMS
2:   candidates := {}
3:   candidate  $c_1$  := replace all wildcards with relaxed
4:   candidates +=  $c_1$ 
5:   results := {}
6:   while candidates is not empty do
7:     Candidate  $c$  := candidates.pop()
8:     Model-check with  $c$  and yield a cycle  $l$ 
9:     if  $l == \text{NULL}$  then
10:       results +=  $c$ 
11:     else
12:       STRENGTHENPARAM( $l$ ,  $c$ , candidates)
13:     end if
14:   end while
15:   return results
16: end function
17: procedure STRENGTHENPARAM( $c$ ,  $p$ , candidates)
18:   while  $\exists$  a pattern  $p$  in cycle  $c$  do
19:     possible_fixes := strengthen  $c$  by pattern  $p$ 
20:     candidates += possible_fixes
21:   end while
22: end procedure

```

Figure 3. Algorithm for Searching All Possible Parameters