Exploring Location Consistency Using Alloy

Mana Taghdiri

May 15, 2002

Abstract

Location Consistency(LC), as defined in [1], is a memory consistency model not based on memory coherence property which is considered as a natural property in almost all other existing memory models. Thus, LC provides a simpler to implement, more scalable and more efficient memory model. I use Alloy to model Location Consistency. The Alloy Analyzer is used to show that LC is strictly weaker than SC(Sequential Consistency), while still equivalent to it for the common case of parallel programs that have no data races. Also I make a model of LC cache coherence protocol proposed in [1] and show that this protocol obeys LC memory model.

1 Background and Motivation

In a shared memory multiprocessor architecture, a *memory model* specifies the values that may be returned by the memory in response to instructions issued by a program.

The first shared memory systems tried to give parallel programmers the same guarantees they had when programming uniprocessors. *Sequential Consistency(SC)* [2], as one of the earliest memory models, requires that the result of any execution of the program be the same as if the operations of all the processors were executed in some sequential order and the operations of each individual processor appear in the order specified by the program. This model is relatively easy to use by programmers, but it has been observed that providing such a strong memory model creates a huge coherence overhead, slowing down the parallel application.

To solve this problem, many relaxed memory models have been proposed which have behaviors different from the traditional uniprocessor one. But there's always a trade-off between providing performance and providing ease of programming for distributed/parallel applications because understanding the behavior of a highly concurrent system is never easy.

Yet, almost all memory models rely on *memory coherence* assumption [3] which basically says "all writes to the same location are serialized in some order and are performed in that order with respect to any perocessor." This assumption makes the model less scalable and efficient while hard to implement.

Location Consistency(LC) [1], is a memory model not based on memory coherence assumption. Thus, the values returned by read operations in this model, are sometimes hard to analyze, much different from the results of program execution on other models. So LC is not easy to use by ordinary programmers, but it allows a high amount of concurrency in programs which results in better performance and efficiency. Cache coherence protocol providing LC does not need to enforce single write ownership of memory locations, so is much simpler than other proposed protocols.

2 Summary and Evaluation

A simplified version of LC, handles four memory instructions: $read(p_i, l)$, $write(p_i, l, v)$, $acquire(p_i, l)$ and $release(p_i, l)$ while p_i is the processor executing the instruction, l shows the location and v is an arbitrary value. *acquire − release* are used for data synchronization when the processor p_i needs exclusive access to a shared location l. *acquire* instruction should be used at the start of the critical section and *release* is used at the end of that. It is assumed that each *acquire/release* pair of operations performed on the same memory location is issued by the same processor.

In LC, the state of a memory location is viewed as a partial order (rather than a total order) of write and synchronization operations. In general, each memory location has a separate partial order which captures the sequencing constraint of the memory write and synchronization instructions performed on the location. The partial order of a location imposes that all write and synchronization operations on that location, issued by the same processor, be totally ordered and also each *acquire* instruction on that location follows the most recent release operation (regardless of the processor performing that).

According to the partial order defined for a location, the value returned by a *read* on that location can be any value written by a *write* instruction not preceding it (in the partial order) or written by the most recent *write* preceding it.

Because of the structured nature of LC and its completely different behavior in compare with other memory models, I decided to model it using Alloy. The basic signatures of the Alloy model are as follows:

```
sig Location {
        StateOps : Tick -> StateOperations,
        PartOrder : Tick -> StateOperations -> StateOperations}

sig Operation {
        P : Processor,
        T : Tick,
        L : Location}

disj sig ReadOp extends Operation {
        Val : Value}
disj sig StateOperations extends Operation {}
disj sig AcquireOp, ReleaseOp extends StateOperations {}
disj sig WriteOp extends StateOperations {
        V : Value }
```

The complete model is presented in the appendix A. Modelling LC was a different experience for me. It took me some time to find out what a program model is, what a program execution (interleaving of program instructions) is and how it should be modeled. Unlike all my previous models I didn't write any explicit constraints for the atoms generated (or modified) in each state. Instead, I constrained any *existing* atom in my model. I believe this is a more declarative model now.

I tried to check two main claims in the paper. First is to show that LC is strictly weaker than any existing memory models. I decided to use Sequential Consistency because it was just a single constraint over my model's instances and so it wouldn't make the model so huge and thus hardly analyzable in reasonable time. But the generated example is applicable to most memory models. The second claim was to show that LC is equivalent to SC for the common class of programs which don't have data races which proves the usefulness of LC. This was true in the scope Alloy was able to analyze the model.

Then I decided to model LC cache Coherence protocol as well. LC cache protocol is much simpler than existing snoopy or directory-based cache protocols. Since there are no memory-coherence or serialization requirements beyond what is implied by the program's partial order, a cache consistency protocol for the LC model does not need to ensure single ownership of memory locations. But since there's no memory coherence requirement, it should guarantee that a read operation always returns an element legal according to LC memory model. The state transition diagram for this cache protocol is enclosed in Appendix D.

The model is enclosed in appendix B. I abstracted some of the possible features like write buffers. Also the model allows each cache line to be written back to the main memory at *any* time. Although the model seems large, but the Alloy Analyzer was able to check it for a fairly good scope (6 elements on a single location - which is reasonble because different locations are completely independent of each other). In order to show that LC cache coherence protocol obeys LC memory model, I needed to show any value returned in a *read* instruction in the cache protocol is actually a legal value according to the memory model. But this wasn't true at first try. Because since *read* operations are not considered as *state operations*[1], the state of a cache *after* a *read* is not well-defined. I should emphasize that although in the cache protocol the behavior of the system is well-defined after a *read* operation, I couldn't inforce that as a constraint over my model, because I was to *check* that. So I decided to restate the assertion as there's only one read in the system and whenever it happens it returns a correct value. This doesn't decrease generality, because by definition reads shouldn't change the location's state and so there is no need to have more than one read operation. Then Alloy showed that LC cache coherence protocol provides LC memory model.

3 Lessons Learnt

LC has interesting semantics and it can generate unexpected values in response to *read* instructions. But Alloy could produce very simple counterexamples to my assertions. I used Alloy examples to think of more complicated cases and so refine my assertions. It took me some time to write the assertions which could have meaningful counterexamples. One of the counterexamples is enclosed in appendix C.

Visualization of the instances was a time-consuming task. I couldn't produce the output which was really easy to follow by eye. The main problem was with *StateOps* relation. While projecting over *Tick* relation, I wanted to show only the operations involved in that specific tick, but seemed impossible. so I decided to hack Alloy's output manually and for example I deleted some nodes one by one in Alloy visualized output before I could print out the result. I think what is presented in appendix C is easy to see but I'm not sure if it is the best visualization result I could get.

One of the reasons I selected this relatively big system to model with Alloy was to rethink of modelling in Alloy. While I got those "No solutions found" messages, I tried to be more careful to find out if the model is overconstrained or not. And I saved all my models step by step to use in my overconstrainedness project. It was sometimes really hard to find out what was going wrong with the constraints, taking much more time than expected. In modelling a system like LC, it is very important not to make anything more constrained than it must be, because it will eliminate any chance of finding bugs with the system. So I'm now more determined to think about how Alloy can help people to create perfect models without spending such a long time debugging the model itself. It undoubtedly should be able to produce more useful messages than "No Solution Found!".

References

[1] G. R. Gao and V. Sarkar, *Location Consistency - A New Memory Model and Cache Consistency Protocol*, IEEE Trans. Computers, Vol. 49, No. 8, PP. 798-813, Aug. 2000.

[2] L. Lamport, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Trans. Computers, Vol. 28, No. 9, PP. 690-691, Sept. 1979.

[1]State Operations are those operations which may change the state of a location, which are only *write*, *acquire* and *release* according to the definition.

[3] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy, *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*, Proc. 17th Ann. Int'l symp. Computer Architecture, PP. 15-26, May 1990.