Modeling Memory Coherency during concurrent/simultaneous accesses

Subramoni Parameswaran
2101 logic drive
San Jose, CA 95124
subramo@xilinx.com

***Abstract*- The presence of multiple actors that can concurrently modify and read memory complicates the problem of modeling and verifying memory coherency. Practically, it may not be feasible to precisely model the expected value of a memory location observed on a read when there are multiple concurrent writers/readers to the same memory location. In this paper, we develop a framework that can predict the multiple potential coherent values that could be observed upon a read when there are concurrent/simultaneous reads and writes to memory. Using the concept of window of uncertainty, we store all potential write values in our memory model and remove values over time that are known incoherent, thereby verifying coherency even in the presence of uncertainty of read outcomes.**

I. INTRODUCTION

In recent years, there has been a significant increase in the number of parallel actors that can concurrently access memory. One of the critical objectives in verification is to ensure the coherency of a memory in the presence of parallel/concurrent writes and reads. Ref. [1] states that a memory is coherent if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address. Ref. [2] talks about the difficulty of precisely defining the most recent write in shared memory multiprocessor systems. The difficulty of predicting the coherent expected value of a memory location arises when there are multiple actors capable of writing and reading to the memory location either in the same cycle or across multiple different cycles. When multiple writes are outstanding and we issue a read in parallel, it may not be feasible to predict the precise value returned by the read. While architectural use cases may eliminate the issue of unpredictability by temporally or spatially isolating simultaneous accesses, ideally, we would like to verify the design in the presence of temporally and spatially concurrent accesses. The problem is further complicated by features like partial writes, write merging and write/read sizing.

As a practical matter, we may not be able to precisely predict the most recent write operation. Fig. 1 shows the problem of determining the winner between two simultaneously outstanding writes to the same memory location when there is a parallel read to that same location. As shown in Fig.1, the issue of the write to the design on its external interface is the point in time at which the write1 "starts". The design updates the corresponding memory location at some subsequent point (write1 is "observed" by the design) and subsequently, write1 gets "acknowledged". "Acknowledgment" or the completion of a write by the design can be defined as the earliest of:

1) The response for a write is received on an interface external to the design.

2) The unique write value is visible to a subsequent read response.

In contrast to writes, "Acknowledgment" for a read in the context of this paper is assumed to only occur when the read response is received on an interface external to the design. We note that if the architecturally defined worst-case latency (if applicable) of a write or read has elapsed and we have not received the response on an interface external to the design, it is automatically a design implementation error. This should be checked independently to ensure correctness of the design.

As shown in Fig. 1, while write1 is being processed by the design, write2 starts and the design begins processing it. If a read were to start in parallel while these two writes were outstanding, the read could coherently return either the value written by write1 or the value written by write2. Without either whitebox methods of observing internal design variables, or, without cycle accurate modeling of the design, it is impossible to predict whether the read is required to return the write1 value or the write2 value. In this case, either value is perfectly permissible and coherent as returned by the read.

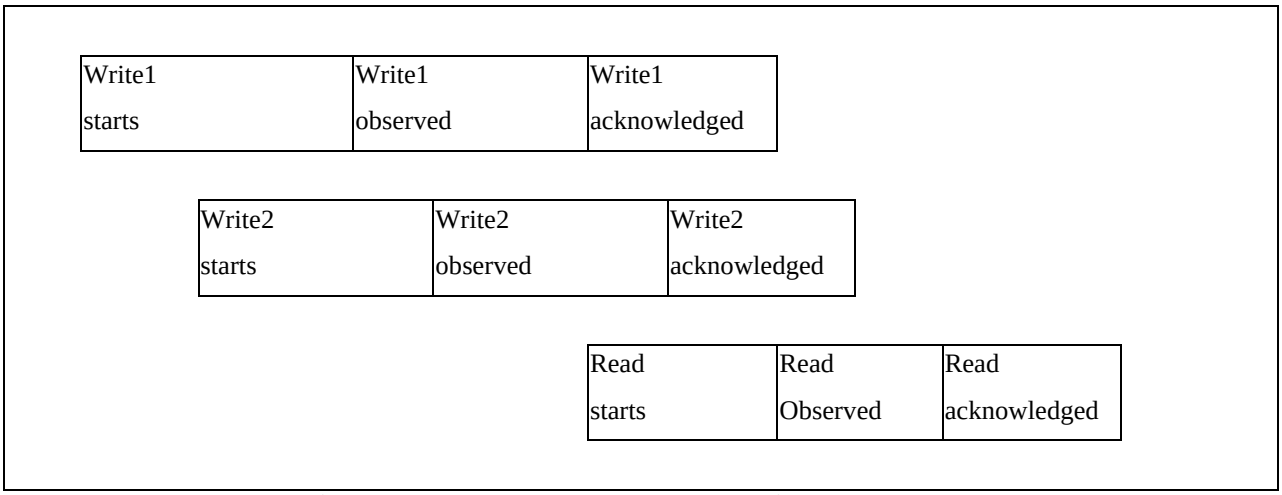

Figure 1. Problem of predicting the observed read value in the presence of two simultaneously outstanding writes

II. COHERENCY MODEL

To address the problem of modeling coherency, we develop a concept called the Window of Uncertainty (WoU), which allows us to model multiple potential write values for the purpose of coherency checks for a read value in the presence of parallel/concurrent stimulus. Uncertainty refers to the potential values introduced by multiple writes that could be seen by a read that is issued after the writes are acknowledged. Even when a specific write is acknowledged, if there are other writes outstanding, the outcome for a subsequent read is still uncertain because its observed data value could be one of the other outstanding writes and we need to account for this. The Window of Uncertainty is framed by the start of uncertainty and the end of uncertainty.

A. Start of uncertainty

When a write becomes outstanding, it creates a window of uncertainty, which is a time window where it's value could be the potential coherent value of a subsequent read to its memory destination. The write introduces uncertainty of outcomes for subsequent reads when it is issued - For example, the read may return the previous written value or the value of the currently issued write. Therefore, the start of uncertainty is the time of issue of a write.

B. End of uncertainty

The end of uncertainty is more complicated to determine precisely. However, the window of uncertainty of a write terminates when it is known that either other writes from the same actor or from different actors must necessarily supersede the expected value in the memory location.

B1. End of uncertainty for two writes from the same actor

Let us illustrate this point by considering the case of two writes from a single actor as shown in Fig. 2. Because write1 and write2 are simultaneously outstanding, even at the point in time where write1 is acknowledged, it is possible for a subsequent read to return write2 as the value. In the single actor case, for a write, the end of uncertainty for subsequent reads occurs at the point of acknowledgment of the subsequent write i.e. write2. At that point, assuming no further writes are issued, any read issued after the write2 acknowledgment is guaranteed to *not* return the write1 value. In other words, given that write2 is issued after write1, and write1 and write2 are from the same actor, coherency rules dictate that write2 should supersede the prior write1 value after it is acknowledged. As a result, this causes an end to the uncertainty created by the issue of write1. As shown in Fig. 2, write1 can no longer be the result of the read issued after write2 is acknowledged.

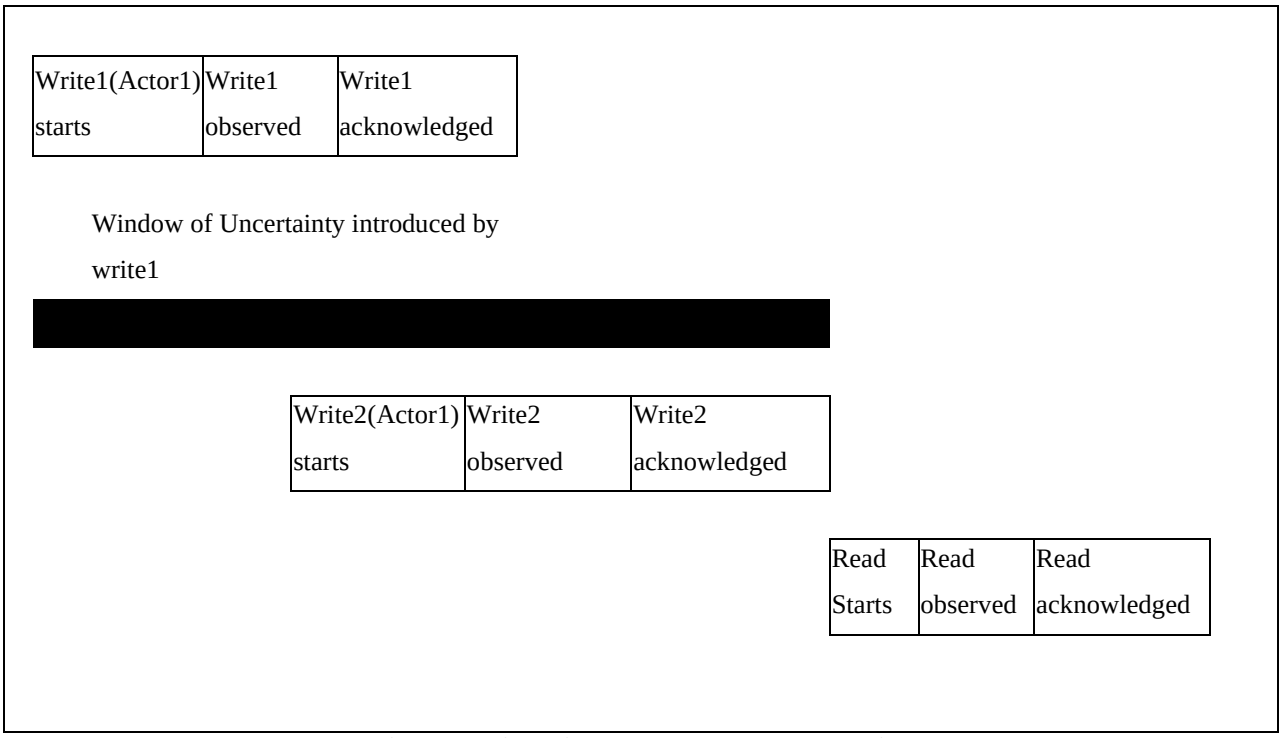

Figure 2. Window of uncertainty for the first write when there are two writes from the same actor

B2. End of uncertainty for two writes from different actors

Now let us consider the case of writes from different actors. Note that for the case of writes from different actors, if two writes are simultaneously outstanding (as opposed to one getting issued after the other is acknowledged), either the first write or the second write issued could very well be the final value that is stored in memory after both writes are acknowledged because the write latency could be very different for the two actors.

This is shown in Fig. 3a and Fig 3b. In both figures, write1 is issued ahead of write 2. However, in Fig. 3a, write1 is observed by the design *prior* to write2 being observed. As a result, the write2 value is the final value in memory. However, in Fig. 3b, write1 is observed by the design *after* write2 is observed. This could happen because write1 has a long latency and write2 has a short latency given that they are issued from two different actors. As a result, the write1 value is the final value in memory. In summary, for writes simultaneously issued from two different actors, without observation of internal design variables, there is uncertainty of outcomes. We model this uncertainty of outcomes for two different actors by simply *not* terminating the WoU of write1 when write2 gets acknowledged because it is not possible to do so. This is shown in both Fig. 3a and Fig. 3b. Stated differently, given our inability to distinguish the order in which write1/write2 will be observed by the design (Fig. 3a vs. Fig. 3b), we simply presume uncertainty of outcomes for write1 both in the case of Fig. 3a and in the case of Fig. 3b.

However, in the case of two writes issued from two different actors, the end of uncertainty *will* occur if a subsequent write that is issued *after* the acknowledgment of the current write is itself acknowledged. At that point, the subsequent write, even if from a different actor, is guaranteed to supersede the write from the current actor.

This is shown in Fig. 4 - if write1 is issued and acknowledged, and subsequently write2 is issued and acknowledged, coherency rules dictate that the memory location must store the write2 value despite write1 and write2 being issued from different actors.

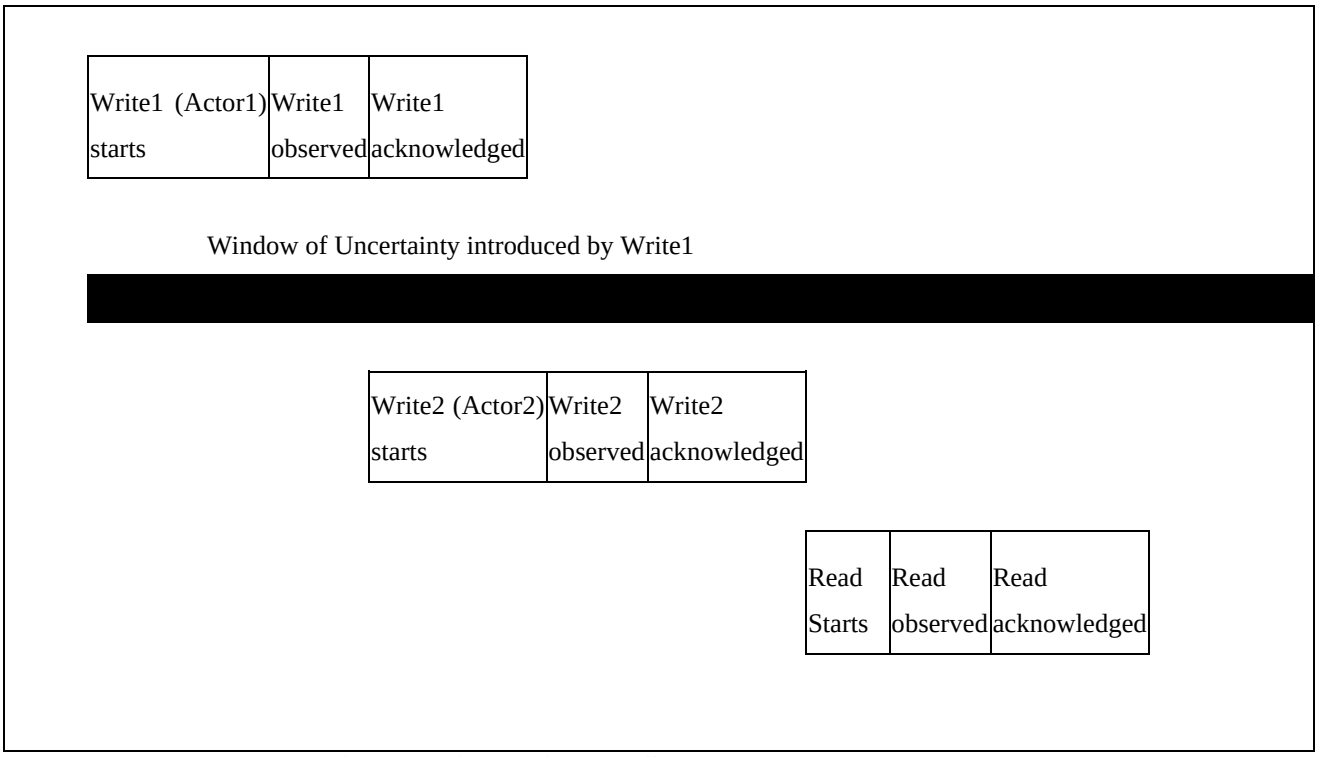

Figure 3a. Window of uncertainty for writes from two different actors (Write1 has shorter latency than write2)

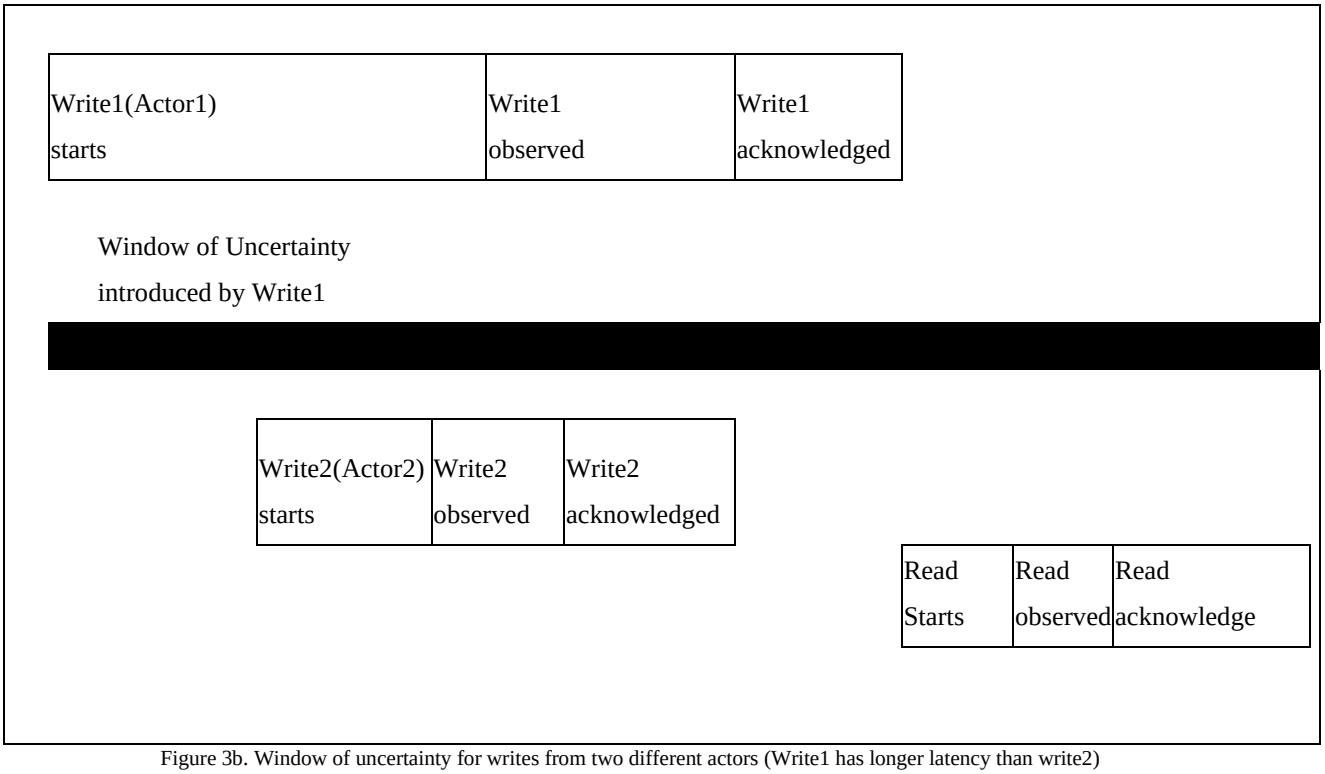

Figure 3b. Window of uncertainty for writes from two different actors (Write1 has longer latency than write2)

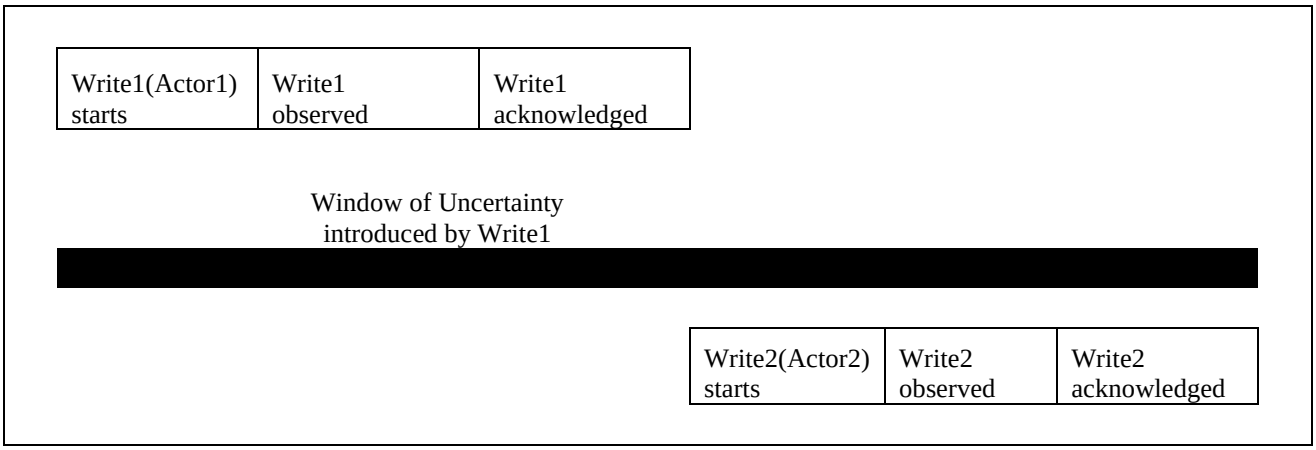

Figure 4. Window of uncertainty for writes from different actors (Write2 issued after write1 acknowledgment)

III. MODEL IMPLEMENTATION

The steps involved in the specific implementation of the model are the following:

1) We choose the granularity of modeling the memory location based on the smallest write size (or granularity).
2) We store multiple data values that might be the "winner" per memory location including their issue timestamp, and their acknowledgment timestamp.
3) We order the potential data values per memory location in the Potential Values queue based on the time of issue of the writes.
4) We can then use the window of uncertainty applicable across same/multiple actors to remove superseded entries to derive all potential write values that are coherent at that memory location for future reads.

Fig. 5 shows the multiple potential write values that are stored in the Potential Values queue of the coherent memory model per memory location for the purpose of modeling values returned by subsequent reads.

C. Removal of superseded entries

The first rule states that a write from the same actor is superseded when a subsequent write from the same actor is acknowledged. When a write is acknowledged, we search each memory location for the immediately prior write from the same actor and we delete that entry from the Potential Values queue for that memory location.

Fig. 6 illustrates this with an example. Write10 and write13 are both issued from actor1. When write13 is acknowledged, write10's window of uncertainty terminates, and it can be removed as a potential match for that memory location. This is true even though write10 and write13 are simultaneously active when write13 is issued -

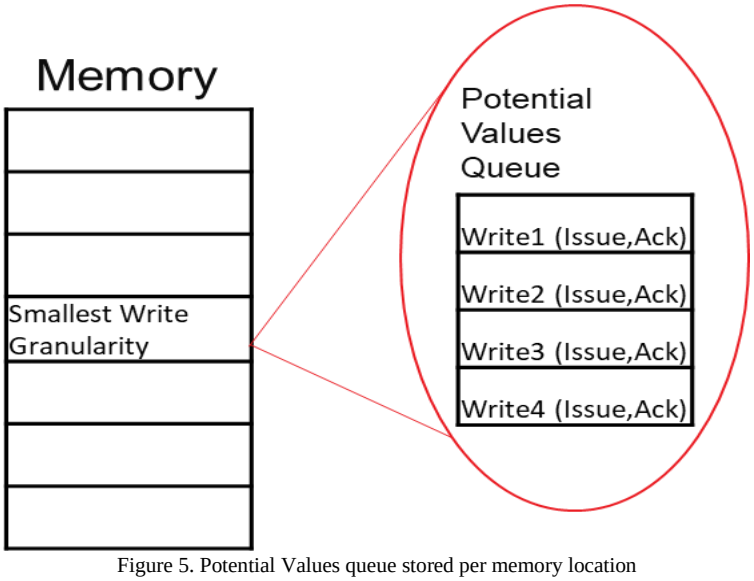

Figure 5. Potential Values queue stored per memory location

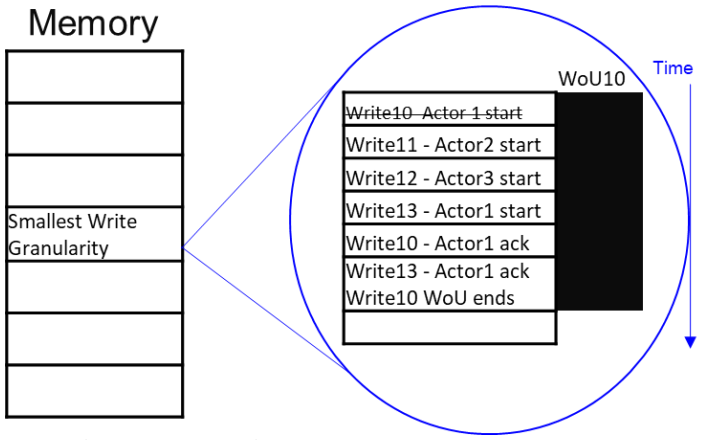

Figure 6. Removal of superseded writes from the same actor based on memory location activity over time

(write10 still has not received its acknowledge when write13 is issued). This rule can also be applied in the presence of active writes from other actors because it relies only on sequential processing of writes from the same actor.

The second rule states that if a write is issued, and, after its acknowledgment, a second write from any actor is issued and the second write has also been acknowledged, then the first write can be deleted from the queue. When a write is acknowledged, if there are prior writes from a different actor present as a potential match for that memory location that were acknowledged before the issue of the current write, the second rule can be used to delete entries from the queue of potential matches for that memory location. This is the reason that the Potential Values queue needs to store the time of issue and the time of acknowledgment of all writes as shown previously in Fig. 5.

Fig. 7 illustrates this with an example. Write1 is issued from actor1 and write2 is issued from actor2. At the time write2 is acknowledged, we look through the Potential Values queue to see if there is a write issued from *any* actor that was acknowledged before the current write was issued. In this case, write1 was issued and acknowledged before write2 was issued and write2 has now completed. So, we can safely consider write1 to be superseded by write2 and we can remove it from the queue of potential matches for that memory location.

D. Creating the expected values for a read from the model

A read that is issued could return one of the potential values from the Potential Values queue at its time of issue or any write that is issued while the read is outstanding. We could only look at the Potential Values queue at the time the read is acknowledged to generate all potential coherent values for the read. However, this would be incomplete in terms of potential coherent values because we could miss writes that were *superseded* between the time the read is issued and its acknowledgment. Even if a write was superseded while the read was active, depending on the timing of the read, the read could potentially pick that write value as the coherent value returned.

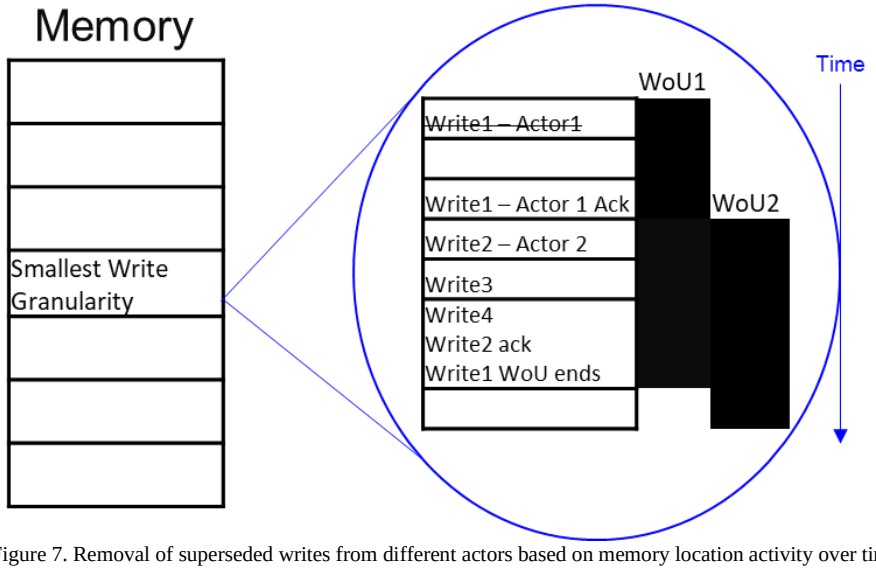

Figure 7. Removal of superseded writes from different actors based on memory location activity over time

To avoid this problem, on a per read basis, we need to keep track of writes that were superseded after the time of issue of the read (until its acknowledgment). To do this, we need to create another queue called the Active Reads queue (Specific Potential Values queue) which, on a per-read basis, keeps track of the writes superseded in the Potential Values queue while the read was outstanding. All other potential write values that were not superseded after the time of issue of the read will be captured in the Potential Values queue at the time of acknowledgment of the read.

Another way to state the same concept is the following. To generate the expected value for a read, we need to read the potential values from the model at the time of issue of the read and continue to append all write values that are issued after the read is issued until the read data returns. This is necessary because at the time of issue of the read, any prior potential values are certainly possible for the read to return as a coherent value. In addition, writes that are simultaneously issued when the read is outstanding could be observed ahead of the read and legally affect the coherent value returned by the read. We solve this problem by:

1) Looking at the Potential Values queue when the read is acknowledged (as opposed to issued).
2) By keeping track of superseded writes in the Active Reads queue while the specific read is outstanding.

Fig.8 shows this in action. The memory location activity across time is shown on the left. The two queues are shown on the right – the Potential Values queue and the Active Reads queue. Write1, write2 and write3 are issued and written into the Potential Values queue, and subsequently read1 is issued. At the time of issue of read1, the Potential Values queue contains all three writes because none of them have been superseded. While read1 is active, because write1 and write2 get acknowledged and they are from the same actor, write1 gets superseded and removed from the Potential Values queue. Because read1 is active at the time that write1 is superseded, when write1 is superseded and removed from the Potential Values queue, it is inserted as a potential match for read1 into the Active Reads (Specific Potential Values) queue. Subsequently read2 gets issued, write3 gets an acknowledge and as a result, write2 gets superseded. When write2 is removed from the Potential Values queue, both read1 and read2 are active, so both get write2 as a potential match in the Active Reads Queue. Now if read1 or read2 were to get acknowledged, read1 could coherently return as potential values the set {write1, write2, write3}. However, read2 could only coherently return {write2, write3}. Therefore, acceptable read response match values are defined as the union of the values specific to the read in the Active Reads queue (Specific Potential Values queue) and all the values present in the Potential Values queue at the time of acknowledgment of the read.

E. Sub-cycle order of operations

Thus far, we have not defined how the memory model will be built when there are multiple reads or writes that get issued/acknowledged in the same cycle. We now define the order in which we assemble the model in the presence of multiple read/write operations in the same cycle.

1) If any reads are acknowledged, create their specific set of acceptable match values by combining the set of values in the Potential Values queue and the specific set of entries in the Active Reads queue.
    a. Check that the read data returned corresponds to the combined set of potential values.
    b. Optionally, if the acceptable match value set is unique (no duplicated entries) and the specific matched write value is present in the Potential Values queue (not the Active Reads queue), consider the corresponding write to be acknowledged and remove any superseded entries from the Potential Values queue using the first rule or second rule. Add the superseded entries from the Potential Values queue into the Active Reads queue.
    c. Delete the read that was acknowledged from the Active Reads queue.
2) If any writes are acknowledged, remove any entries in the Potential Values queue that were superseded as a result and add them to the Active Reads queue.
3) If any reads are issued, add them to the Active Reads queue.
4) Add any writes issued this cycle to the Potential Values queue.

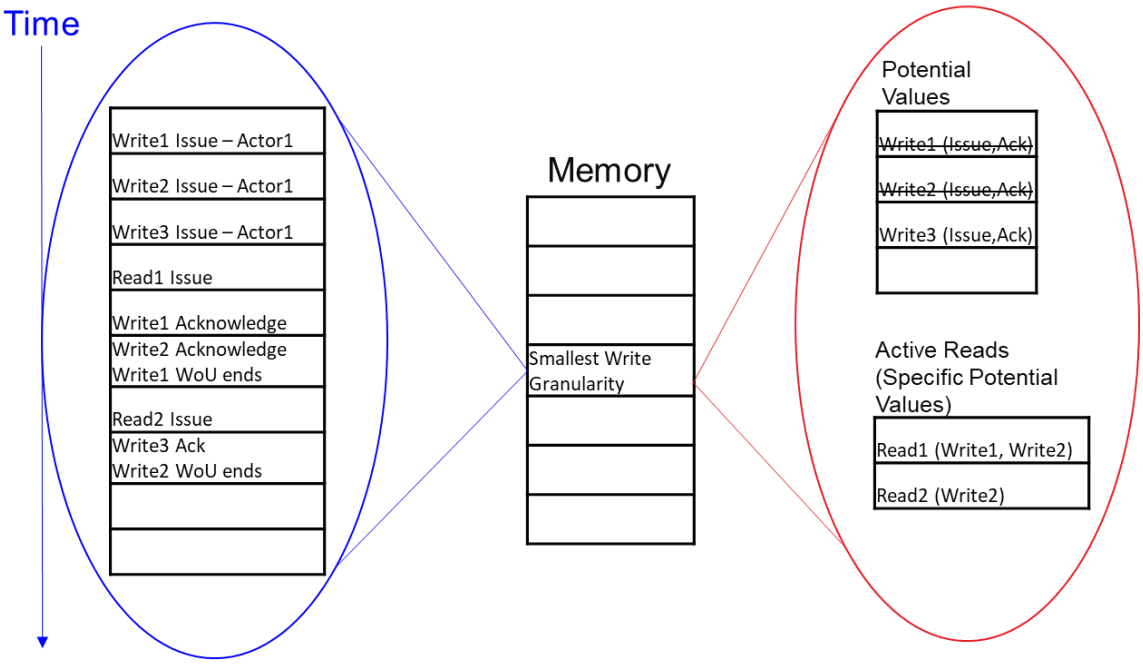

Figure 8. Dealing with writes that are issued while the read is active

F. Adapting the model to check non-concurrent stimulus

Consider the case of "architecturally coherent" stimulus where it is guaranteed by the test writer that there is only one potential coherent value for the read. This is typically done by ensuring that there is never more than one outstanding transaction (write or read) to a specific memory location by temporally and/or spatially separating multiple writes and reads to the same memory location.

The model can be easily adapted for this type of "coherent" stimulus by checking that two writes are never issued in the same cycle and by ensuring that there is never more than one potential value in the potential writes queue (the acknowledgment for all writes should be assumed to happen in the same cycle as the issue). We can also check that there are never any potential values in the Active Reads queue (no writes should be superseded while a read is outstanding). In other words, the model can be easily adapted to deal with stimulus that is architecturally coherent (reads and writes to the same memory location are either spatially or temporally separated) if desired. It can also be used to check the specific value returned by a read in this case because there should only be one value in the potential value queue and no values in the Active Reads queue.

IV. CONCLUSION

We have developed a generic coherent memory model for uncertain read outcomes in the presence of multiple concurrent actors and shown how it can be leveraged to check the coherency of memory outcomes for a read. We have developed two simple rules that can help us model coherent values that can be returned for a read and dealt with the problem of superseded writes while the read is outstanding. Use of this model can support more extensive concurrent verification of memory. Furthermore, this model can be easily adapted to verify non-concurrent (spatially or temporally separated) stimulus by ensuring that there is at most one coherent potential value in each memory location.

REFERENCES

[1] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *PODC '86.*
[2] R. Steinke and G. Nutt, "A unified theory of shared memory consistency," *Journal of the ACM. 51 (5): 800-849.*