BatchSolve: A Divide and Conquer Approach to Solving the Memory Ordering Problem

Debarshi Chatterjee, Ismet Bayraktaroglu, Nikhil Sathe, Kavya Shagrithaya, Siddhanth Dhodhi, Spandan Kachhadiya
Nvidia Corporation
2788 San Thomas Expy
Santa Clara, CA - 95051

***Abstract*- Memory Ordering Problem (MOP) arises frequently in unit and integration level testbenches (TBs) where Bus Function Models (BFMs) drive Memory Operations (MemOps) on various interfaces of the Design Under Test (DUT). Each MemOp has a set of attributes such as, MemOpType (Reads/Writes/Atomics/Barrier), destination MemType (System/Video Memory), SourceType (CPU core, GPU SM), address, length etc. Certain MemOps like reads, non-posted writes, certain atomics and barriers get response (or acknowledgment) back. These responses/acks, their attributes and timings will be collectively termed as Resp-Attributes. Given the issue order of MemOps and all information about the MemOp-Attributes and Response-Attributes as observed on the DUT boundary, the problem is to find whether there exists a Global Order (GO) of the MemOps such that: 1) The read data observed matches the last write data in GO to the same address and 2) The order of MemOps in GO satisfies various ordering rules in the Implementational and Architectural Specifications (IAS). Ordering rules can vary greatly across various design units and link connectivity. For example, in a GPU TB when MemOps are going to CPU memory, the ordering rules can vary based on the type of the link (PCIE vs NVLink) over which CPU is connected. MOP is similar to Memory Consistency Problem (MCP) which is known to be NP-complete [1]. Existing techniques modify this problem under reasonable assumptions to make it tractable. We will provide an overview of such techniques in the previous work section. Such techniques either require high development and maintenance cost or are not flexible enough to allow users to write arbitrary ordering rules at a high level. In this paper, we introduce Batch-Solve (BATS) - a low-cost, scalable, portable, and flexible approach to solving MOP. Our formulation allows users to specify ordering rules as high-level SystemVerilog (SV) constraints. We leverage the power of SV constraint solver to determine if there exists a GO that satisfies the user specified high-level ordering constraints and matches the results of an execution. BATS can be easily ported over to another TB (horizontal re-use) and is immune to architectural modifications (vertical re-use across projects). To the best of our knowledge, this is the first SV Solver based approach to solving the MOP.**

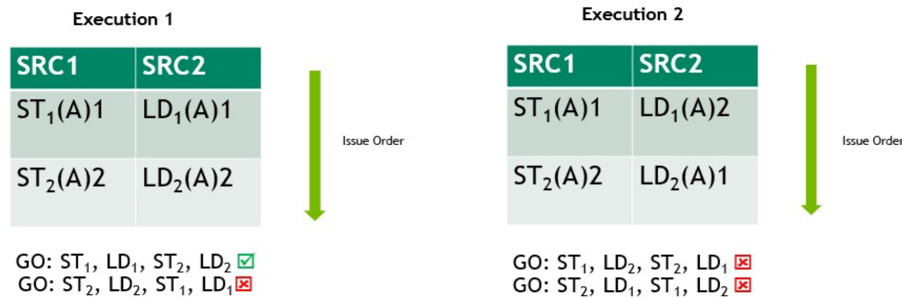


Figure 1. Inferring Global Order based on execution and checking for existence of legal GO

I. Introduction

To understand MOP, let us consider two sources SRC1 and SRC2. Let $LD_n(A)X$ denote a Load to address A that returns data X; $ST_n(A)Y$ denote a store to address A with data Y. For Execution1, in Fig. 1, assume a test where, $ST_1(A)1$, $ST_2(A)2$ was driven by SRC1 (in that order) and $LD_1(A)1$, $LD_2(A)2$ was driven by SRC2 (in that order) and the data return happened to SRC2 in any order. Also assume the initial value of the memory location A was 0. Since LD_1 got the data written by ST_1, we can infer LD_1 was ordered after ST_1 and there was no intervening store to A between them. Similarly, we can infer LD_2 was ordered after ST_2. So, we have two partial orders: (ST_1, LD_1) and (ST_2, LD_2) which could have been permuted arbitrarily. An external observer can therefore infer that the Global Order (GO) imposed by DUT could be either (ST_1, LD_1, ST_2, LD_2) or (ST_2, LD_2, ST_1, LD_1). Note that the external observer cannot be sure about which GO actually happened in the DUT without probing the internal signals of the DUT. Now, let us suppose the IAS specification mandates that DUT follows the property *P* that the loads/stores from each source must appear in GO in the same order as they appear in the issue order. We can observe that out of the two possible inferred GO, only (ST_1, LD_1, ST_2, LD_2) follows the ordering rule *P*. Since there exists a GO (marked in green tick in

the Fig. 1) that satisfies the ordering rule *P*, we will decide the execution 1 to be legal. Next, look at Execution 2 in Fig. 1. In this case, the load data received is swapped and everything else is same. By similar logic as explained before, an external observer would infer that the GO is either (ST_2, LD_1, ST_1, LD_2) or (ST_1, LD_2, ST_2, LD_1). However, in none of these inferred GOs the ordering rule *P* is followed. Since there does not exist a GO which satisfies the ordering rule, we will call this execution to be illegal. The idea here is to catch such ordering rule violations in the design.

Although this example might seem simple, the problem gets complex very quickly once you add the following elements: 1) Increase the number of sources and MemOps from each source 2) Increase the number of properties that the DUT needs to obey 3) Consider the fact that MemOps are not always single-byte, but each could be any length from 1-N, where N is depends on the design 4) Each MemOp can cross the granularity across which ordering is maintained. In these cases, typically we break the MemOps into multiple child transactions at the boundary of the ordering granularity and check the GO of the child transactions (This is explained later) 5) MemOps other than load/stores, like atomics, barriers also have different ordering rules. Not just that, ordering rules for a particular MemOp type might depend on other attributes of the MemOp. For example, ordering of writes on PCIE depend on whether they are strict or relaxed order.

II. Background And Previous Work

In this section we will briefly describe some prior work on this topic. Verifying MCP is a well-researched field with many interesting works done on this topic [2][3][4][5][6][7]. However, there are some key differences between MCP and MOP which we would like to highlight. The first difference is that MOP deals with ordering of MemOps which are much close to the hardware implementation than MCP which typically finds a GO of the assembly instructions. Ordering these memory operations pose a different set of challenges due to the nature of these operations. For example, the problem of ordering of writes with byte enable holes never arise in MCP (More on this later). Another example is reads generated by hardware prefetch which has no notion of program order, but still needs to obey ordering rules in MOP. The second difference is that in MCP we are verifying whether GO is legal from memory model and program order standpoint, whereas in MOP we are verifying lower-level ordering rules of memory operations within a unit or in a subsystem. Such ordering rules ensure not just data consistency but also deadlock avoidance. Typically, such ordering rules vary from unit to unit and are much complex than ordering rules for MCP. For example, refer to PCIE ordering rules [8]. Since violations of these rules are necessary but not sufficient conditions for deadlock and data consistency bugs, early detection of such violations at unit and subsystem level is very important. Due to these differences, all existing work on MCP might not be directly applicable to MOP. Hence, we will focus on approaches that have been proven to deal with the complexities of verifying MOP in the context of complex industrial design. In this paper, we will primarily discuss two interesting prior approaches that have been implemented in Nvidia's verification environment – 1) TSO Tool 2) Point of Serialization Snooping (POSS). We use these approaches as baseline to compare and contrast the pros and cons of the proposed approach.

Total Store Order (TSO) tool [6], which was originally developed to verify MCP of TSO memory model can be applied to solving MOP with some limitations. Hangal et. al. [6] show that if all stores to a specific memory location have unique data, then this problem can be solved by a polynomial time algorithm. The basic idea is to construct a directed graph where each node represents a MemOp. Due to unique data, each load will get data from a unique store (or initial data). One can draw edges from the store to the load (from which the load got its data). Similarly, each ordering rule can be represented as a directed edge in the graph. If the graph constructed in this manner is acyclic then it guarantees that none of the ordering rules are violated. TSO tool is very versatile and provides very good temporal coverage. It is also portable and immune to micro-architectural changes. However, due to unique data requirement, handling random atomic MemOps (e.g., MIN, MAX, Compare-and-Swap) is specially challenging in this method. Due to the same reason, number of single-byte writes in a test are limited to 256 in this method. Moreover, this method requires converting ordering rules into edge rules in the graph. For MOP, the ordering rules are not fixed and vary from unit to unit. As a result, converting these rules into edge rules of the graph can be a non-trivial process. Nevertheless, TSO tool has been used to verify MOP successfully with certain assumptions.

Another common approach used in pre-silicon verification of MOP is Point-of-Serialization Snooping (POSS). In this approach we track every MemOp all the way to the Point-of-Serialization in the design and use that information to obtain the GO. We then check if the GO follows all the ordering rules. A major advantage of this approach is that theoretically it can verify any stimulus with no coverage loss. However, in complex industrial designs, end-to-end transaction tracking requires a lot of effort. Moreover, the tracking code needs to be updated if the design changes. Despite high effort and low portability, POSS remains an important tool in the toolbox for verifying MOP.

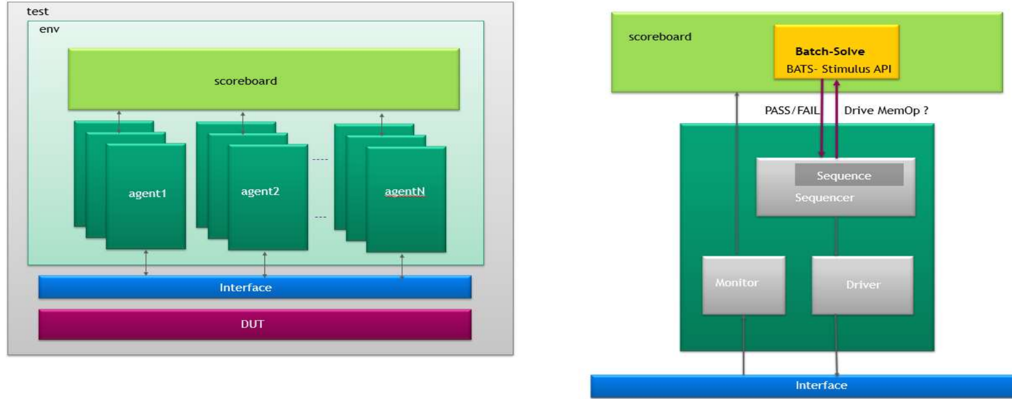


Figure 2. Integration of BATS in standard UVM TB Architecture

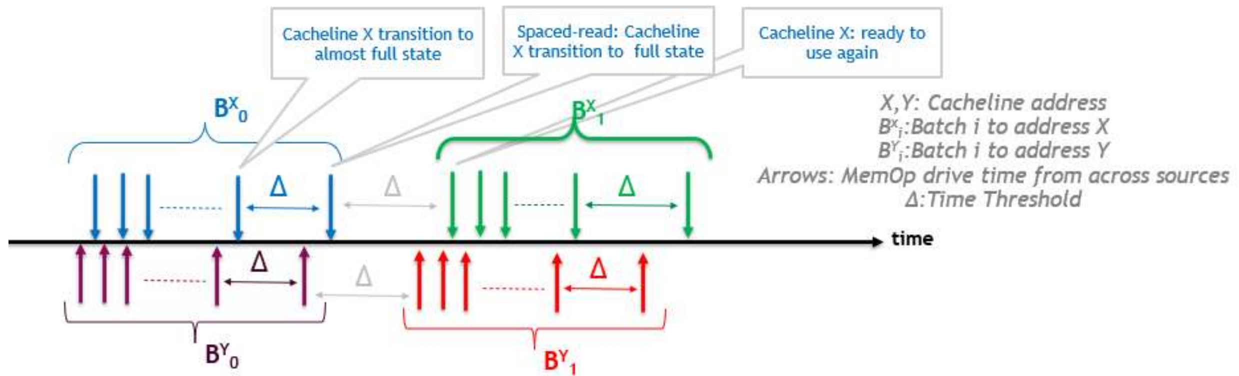


Figure 3. MemOp drive times from multiple sources on the time-line in presence of stimulus batching. Each arrow represent a MemOp drive time on the time-line. Arrows of the same color belong to the same batch.

II. Batch-Solve High Level Overview

In this section, we provide a high-level overview of Batch-Solve (BATS) and how it can be integrated in a standard UVM TB framework. BATS is instantiated in the scoreboard (as show in Fig. 2). During the UVM *run_phase*, sequences within UVM agents communicate with BATS to decide which stimulus can be driven on the DUT (More on this in Section III). Most TBs have monitors on DUT primary inputs and output interfaces which connect to a scoreboard. The scoreboard collects all information of MemOps, MemOp-Attributes and Response-Attributes. This information is fed to *BATS* during the UVM *check_phase*. BATS then uses an in-built mathematical formulation of this problem that can determine if there exists a GO for the specific test execution based on the ordering constraints provided. The formulation allows this problem to be solved by the SV Constraint Randomizer. Details of the mathematical formulation are discussed in Section V. Ordering of atomics and barriers require special consideration and are discussed in Section VI and VII.

III. Batch-Solve Stimulus Batching Mechanism

Finding whether a legal GO exists or not is an NP-complete problem. The runtime for SV solver which checks for the existence of GO scales exponentially as number of requests going to an address increase. Simply put, you can think that in the worst-case the solver needs to search for all possible permutations to find if a legal GO exists or not (although this is not exactly what happens inside the solver). To circumvent this problem, *BATS* provides a stimulus batching mechanism. The stimulus batching mechanism makes sure that only a fixed number of MemOps to the same address (from same or different agents) can race against each other in a window of simulation time. We will call this fixed number of MemOps as the batch-size (B_s). Since all UVM agents in TB communicate through a central BATS object in the scoreboard, BATS can ensure this behavior by simply suspending the usage of an address once it has

been used B_s times. Please note that this scheme does not limit the overall number of requests to a specific address in a test. We can have arbitrarily large number of requests to an address.

To implement stimulus batching, *BATS* provides a *Stimulus-API* which can be called from a UVM sequence (Fig. 2). The UVM sequence queries the *Stimulus-API* whether a MemOp is legal to drive on to the DUT. If *Stimulus-API* returns pass, sequence will drive it off. If it returns fail, sequence will re-randomize the MemOp and query the *Stimulus-API* again. BATS stimulus batching code will count the number of MemOps going to each cacheline address. There will be no restriction to the timing of first B_s MemOps to any cacheline address, where B_s denotes the batch-size. Once the request count for a particular cacheline address reach B_s-1, the cacheline is said to be in almost-full state and MemOps to the same cacheline will be suspended for a certain simulation time Δ. In this window, any attempted request sent to the cache-line by the sequence will be denied by BATS *Stimulus-API* as fail. During this time, the sequence can pick any other cacheline address which is not suspended. Once the simulation time advances by Δ, any request to the almost-full cache-line will get conditional-pass from BATS *Stimulus-API*. This means that the sequence will need to morph the request type to full-cacheline read before sending it out. We will call this read as a *spaced-read*. Once the *spaced-read* is sent out, the cacheline moves to full state. In full state, MemOps to the same cacheline will be again suspended for a certain simulation time Δ, after which the cacheline is ready for MemOps to be sent again. This is demonstrated in Fig. 3. Details of this scheme is presented in the pseudo-code in Algorithm 1. Fig. 4. shows a sample stimulus batching across 2 interfaces SRC1 and SRC2 with MemOps going to same cacheline. The time Δ is determined empirically and is large enough to make sure that there is no racing of MemOps across different batches to the same cacheline.

But why do we need to make sure there is a *spaced-read* at the end of each batch? This is needed to find the final memory value of the cacheline after execution of a batch. This value will be used as initial memory value for the subsequent batch to the same cacheline. To understand this, let us suppose that we have all writes in a batch (all from different sources) and no spaced read at the end. In this case, any ordering of writes will be a legal ordering. From the solver output we will not know the final value of the memory after the execution of the batch. By putting a *spaced-read* at the end of the batch, we make sure that the last read in a batch never races with prior MemOps in the same batch. The value returned by the read is used as the initial memory value for the subsequent batch. This makes batches totally independent of one another from the solver perspective. A spaced-write at the end of each batch would serve the same purpose, but a spaced-read puts more restriction on the GO of other transactions in the batch. One more point to be noted here is that we are doing stimulus batching at cacheline granularity since we assumed that the design does not allow MemOps to crossover cacheline. If it does, then we need to do stimulus batching at a granularity level which the design does not allow MemOps to cross.

IV. Checker Batching Mechanism

To explain how checker batching works let us take an example. We will assume a hypothetical design where MemOps cannot cross the 64B cacheline boundaries but are ordered at 16B sector granularity. This means, based on the start address and length, each MemOps can be broken down into 1-4 child-MemOps depending on how many sectors it spawns. For example, in Fig. 4 Wr1 start address is aligned to cacheline sector0 and it has length=48B (spawning 3 sectors). Hence it is split into Wr1-C0, Wr1-C1, Wr-C2. Each child is assigned a Unique IDentintifier (UID) by the scoreboard. Fig. 4 shows the data collected in scoreboard for all MemOps going to a specific cacheline address. Each row is a MemOp, and each column has information regarding issue time, SRC and various other MemOp attributes. BATS checking happens in the UVM *check_phase* after simulation has completed. There are 2 parts to the BATS checker. The first part is to group the child MemOps into per-sector batches. The second part is to feed each batch to the solver to check the existence of a legal GO. In this section we are going to discuss the first part. Fig. 5. Shows the first sector0-batch corresponding to stimulus in Fig. 4. As is evident from Fig. 4 and Fig. 5, the checker cannot just select first Bs (=10) requests from column 4 of Fig. 4. to form the first sector0 batch. If it does, Wr7-C0 will be included in the Batch0, which is clearly not the case. This happens because *stimulus-API* makes sure first B_s requests to a cacheline forms the first batch, but not all those requests are guaranteed to touch a particular sector. Hence the number of MemOps in the sector level batch is not always B_s but is guaranteed to be less than or equal to B_s. To keep stimulus and checker code independent, checker needs to figure out which child to include in a batch without any inputs from stimulus code. The checker identifies all spaced read children using Algorithm 2. It then splits these child transactions (sorted in issue order) into batches at spaced-read boundary. Each sector-level batch is then fed to the solver. The solver will determine if there exists a GO that satisfies all the ordering rules. If a GO exists, it will print out the GO, as shown in Fig. 6. If the solver finds the GO solution to be infeasible, then there is a possible ordering bug which needs to be investigated. This process is repeated for all batches going to all sectors.

Issue-Time	SRC	MemOp	Sector-0	Sector-1	Sector-2	Sector-3	Batch-num
T1	SRC1	Wr1	Wr1-C0	Wr1-C1	Wr1-C2		**0**
T2	SRC1	Rd1	Rd1-C0	Rd1-C1			**0**
T3	SRC2	Wr2	Wr2-C0	Wr2-C1	Wr2-C2	Wr2-C3	**0**
T4	SRC2	Wr3		Wr3-C0	Wr3-C1	Wr3-C2	**0**
T5	SRC2	Rd2	Rd2-C0	Rd2-C1	Rd2-C2	Rd2-C3	**0**
T6	SRC1	Rd3	Rd3-C0	Rd3-C1	Rd3-C2	Rd3-C3	**0**
T7	SRC1	Wr4	Wr4-C0	Wr4-C1	Wr4-C2		**0**
T8	SRC2	Wr5	Wr5-C0	Wr5-C1	Wr5-C2	Wr5-C3	**0**
T9	SRC2	Rd4	Rd4-C0	Rd4-C1	Rd4-C2		**0**
T10> T9 + Δ	SRC1	Rd5	Rd5-C0	Rd5-C1	Rd5-C2	Rd5-C3	**0**
T11 > T10 + Δ	SRC1	Wr6		Wr6-C0	Wr6-C1	Wr6-C2	**1**
T12	SRC2	Wr7	Wr7-C0	Wr7-C1			**1**

Spaced Read

Figure 4. MemOps going to same cacheline address in issue order. MemOps split into children and batched. B_s=10

Issue-Time	SRC	UID	MemOp	Sector-0	Read Data Rcvd		Byte Enable		Write Data	
					Byte-0	Byte-1	Byte0	Byte1	Byte0	Byte1
0	-		Dummy Init Wr				1	1	I1	I2
T1	SRC1	1	Wr1	Wr1-C0			1	1	X1	X2
T2	SRC1	2	Rd1	Rd1-C0	X3	X4	1	1		
T3	SRC2	3	Wr2	Wr2-C0			1	1	X3	X4
T5	SRC2	4	Rd2	Rd2-C0	X3	X4	1	1		
T6	SRC1	5	Rd3	Rd3-C0	X7	X4	1	1		
T7	SRC1	6	Wr4	Wr4-C0			1	1	X5	X6
T8	SRC2	7	Wr5	Wr5-C0			1	0	X7	X8
T9	SRC2	8	Rd4	Rd4-C0	X5	X6	1	1		
T10	SRC1	9	Rd5	Rd5-C0	X5	X6	1	1		

MemOp at issued at T4 on SRC2 is omitted because it does not have a child to sector0 for which ordering is being tested

Figure 5. Per-sector batch of child MemOps (Batch0 - sector0) from Fig. 4. This table is a sample input to BATS-solver. For simplicity, table shows each MemOp to have 2B instead of 16B per sector.

Issue-Time	SRC	UID	GO	Sector-0	Read Data Rcvd		Byte Enable		Write Data	
					Byte-0	Byte-1	Byte0	Byte1	Byte0	Byte1
0	-		Dummy Init Wr				1	1	I1	I2
T1	SRC1	1	Wr1	Wr1-C0			1	1	X1	X2
T3	SRC2	3	Wr2	Wr2-C0			1	1	X3	X4
T2	SRC1	2	Rd1	Rd1-C0	X3	X4	1	1		
T5	SRC2	4	Rd2	Rd2-C0	X3	X4	1	1		
T8	SRC2	7	Wr5	Wr5-C0			1	0	X7	X8
T6	SRC1	5	Rd4	Rd4-C0	X7	X4	1	1		
T7	SRC1	6	Wr4	Wr4-C0			1	1	X5	X6
T9	SRC2	8	Rd4	Rd4-C0	X5	X6				
T10	SRC1	9	Rd5	Rd5-C0	X5	X6	1	1		

Notice in UID column MemOps from same SRC appears in issue order

Rd1 inferred to be ordered after Wr2

Rd4 inferred to be ordered after Wr2 and Wr5

Figure 6. BATS-Solver output for input in Fig 5. Rows are MemOps in inferred GO subject to a sample ordering rule: MemOps from same source must appear in GO in the same order as they appear in issue order

V. MATHEMATICAL FORMULATION

This section explains the mathematical formulation which allows SV solver to check if a legal Global Order (GO) exists subject to the ordering rules. To explain the mathematical formulation, visualize each MemOp as a row in an input-2D-matrix (***I***). The elements in columns of ***I*** denote the MemOp-Attributes. We will try to find a random permutation of the MemOps (rows) that gives us the GO. Permuting the rows of the ***I*** to generate a random Perm-2D-matrix (***P***) is straightforward. We can create a random permutation of N-integers using SV unique keyword, and then permute the rows of ***I*** based on that permutation, to generate ***P***. We can then write constraints to make sure rows of ***P*** are in GO. If all the MemOps were single byte (or fully overlapping with no byte_enable holes in the writes), then writing such constraint would be trivial. We would simply write a foreach constraint on every consecutive row of ***P***. The constraints would make sure if ROW_i of ***P*** is a read, then it's received data must be same as the data in ROW_{i-1} of ***P***. This just means that every read gets the same data from the last MemOp in the GO.

The complication in writing constraints for GO arises when we consider that MemOps are not always single-byte and there could be non-overlapping reads and writes, with *byte_enable* holes in writes. In such cases, a read in ROW_i of ***P*** may get one byte from the write in ROW_{i-j} and another byte from the write in ROW_{i-k}, with *j!=k*. For example, the GO shown in Fig. 6, Rd4 gets Byte0 from Wr5 and Byte1 from Wr2. This happens because Wr5 is a partial-write with *byte_enable=0* for Byte1. As a rule, if rows of ***P*** are in GO, then read in ROW_i would get the byte-data from the write in ROW_{i-j} if write in ROW_{i-j} has *byte_enable=1* and there are no writes between ROW_i and ROW_{i-j} that have *byte_enable=1* for that specific byte. We would encourage readers to try and write SV constraints to implement this rule, to better appreciate the fact that this is not straightforward. One might be tempted to solve and find GO per-byte and then somehow merge these. Question is how do we merge the per-byte ordering to find the final GO? We found several working solutions to this problem – some of the solutions are more complex than the others and not all solutions are easily extendible to handle atomics (which will be discussed later). In this paper, we share a simple and elegant solution to this problem which eases debug and is also extendable to atomics, barriers, and other instruction.

The basic idea is to use a dummy random 2D-matrix – let's call it Mem-2D-matrix (***M***). ROW_i of ***M*** would mimic the value of memory after the execution of MemOp in ROW_i of ***P***. Now, all we need to do is write constraints between

rows of ***P*** and ***M***. There will be 3 constraints – 1) If ROW_i of ***P*** is a read with byte_enable=1, then equate the data received by that read (for that byte) to the corresponding byte in ROW_i of M. 2) If ROW_i of ***P*** is a write with *byte_enable=1*, then equate the data written (for that byte) to the corresponding byte in ROW_i of M 3) If ROW_i of ***P*** is a write with byte_enable=0 or a read, then data does not change between ROW_i and ROW_{i-1} of M. Fig. 7. shows a simplified version of this formulation. The aforementioned constraints are implemented under constraint blocks ***c_read, c_write and c_invariance*** respectively in Fig. 7. A sample ordering rule is implemented under ***c_ordering_constraints.*** Such ordering rules can be changed by the user depending on architectural specifications and ordering rules of a particular design unit. When the SV randomizer is invoked, if no such matrix ***P*** exists subject to ordering rules, then the solver would fail with constraint inconsistency failure indicating there is a possible ordering bug in the design. Here is a link to the solution in EDA playground for readers to play and understand better: *https://www.edaplayground.com/x/rXKN.* In the link, we also provide 2 sample executions – one execution for which a legal GO exists and the other for which legal GO does not exist. The first example of legal execution is the same as Fig. 5. Readers can verify that there are multiple legal GO for this execution and Fig. 6 is just one of them.

VI. ATOMIC HANDLING

The formulation explained in the previous section using dummy rand Mem-2D-matrix (***M***) allows atomics to be seamlessly integrated into it. It is easy to see that if, ROW_i of ***P*** is an atomic, then ROW_i of M (which is data in memory after execution of the atomic) is simply a function of atomic data (which is an attribute in ROW_i of ***P***) and the previous memory data (ROW_{i-1} of M). The only thing we must be careful about here is the use of actual function call from within the constraint. SystemVerilog functions within constraint blocks are called before constraints are solved. Therefore, using function calls to compute the result of the atomic operation could cause the solver not to converge to the solution (even if one exists). We have implemented the function call and commented it in the EDA playground for users to verify this fact. Fortunately, all atomic operations can be expressed as inline constraints without the use of function calls. Sometimes certain atomic operations have *op_size* which specifies at what granularity the atomic operation needs to be applied. There are certain limited combinations based on the *length* and *op_size* of the atomic operations. These combinations can be enumerated out as separate constraints. A perl-preprocessor can help enumerate all possible constraints in such cases. Constraint block ***c_atomics*** in Fig. 7. shows a simplified version of code which imposes GO constraints for atomic operations.

VII. MEMBAR HANDLING

Membar handling requires special consideration in the BATS framework. This is because unlike MemOps like reads, writes and atomics - membars do not have a memory address attribute attached to it. How do we incorporate it in the BATS framework? One naïve approach would be to include each membar in each batch, and then write ordering constraints of other MemOps w.r.t membars. Since batch-size is limited, we think this approach is sub-optimal. Before explaining how Membars are handled in BATS framework, let’s quickly understand it’s use case in a producer-consumer data transfer context.

In the producer-consumer model of thread synchronization, a producer thread (P0) produces a data which it wants to transfer to a consumer thread (P1). The sequence of operations for this is shown in Fig. 8. Let Wr(A)D to denote a write to address A with data D; Rd(A) denote a read to address A and Cmpl(A)Y denote the completion for the read to address A receiving data Y. In Fig. 8. P0 thread writes the data D0 (to be transferred to thread P1) by issuing a Wr(A)D0. P0 then issue a barrier instruction (MemBar). Once P0 receives the MemBar acknowledgment, it then sets a flag to location F by issuing Wr(F)1. The consumer thread P1 keeps polling on the flag location, i.e., issues Rd(F) until it receives Cmp(F)1 and then issues Rd(A) to receive Cmpl(A)D1 as completion. The producer-consumer data transfer is expected to yield D1=D0. Various cases can arise here based on whether threads P0 and P1 are running both on CPU; both on same GPU; one in CPU other in GPU; one in one GPU and the other in a peer GPU etc. To complicate things further, either the data and flag locations could be in CPU or GPUs memory. For all such combinations, the sequence of instructions in P0 and P1 should be such that the underlying ordering rules in the DUT would guarantee that the consumer always sees the producer data. If sequences like this is embedded in random stimulus, then BATS framework needs to detect that and make sure that the ordering rules are obeyed properly so that the producer-consumer data transfer can happen.

Now let’s return to the BATS framework and see how we would verify the ordering rules imposed by MemBar in this framework. This involves a 3-step process: ***Step1****:* For every membar find the set of MemOps which should be ordered before the membar (S_{before}) and the set of MemOps which should be ordered after the Membar (S_{after}). Note that MemOps of different addresses can be in either set. ***Step2***: Remove the membar and encode the information

```
typedef enum bit[1:0] {READ, WRITE, ATOMIC_AND} cmd_type;

class BATS_demo;
  // -----------------------------------------------------------------------------------------------------------
  // Matrix I (Input). Each Row of I is a MemOp. Each Column of I is MemOp attribute. Rows are in issue order.
  // For example, I_cmd_type is a column of I. When a MemOp is issued, it's cmd_type is pushed in I_cmd_type
  cmd_type                    I_cmd_type[$];
  bit    [1:0]                I_be[$];           // byte_enable
  bit    [1:0][7:0]           I_data[$];         // Supporting 2 Bytes per cmd

  // -----------------------------------------------------------------------------------------------------------
  // Matrix P (Row Permuted I). Each Row of P MemOp and Column of P is MemOp attribute. Rows of P are in GO.
  rand  cmd_type              P_cmd_type[$];
  rand  bit       [1:0]       P_be[$];
  rand  bit       [1:0][7:0]  P_data[$];

  // ---------------------------------------------------------------------------------------------------------------------
  // Matrix M (Dummy variable). Each row of M mimics the memory value after execution of the memop in corresponding row of P
  rand  bit       [1:0][7:0]  M[];

  rand  int                  G_order[];  // Will have the final GO.
  string                     SRC[$];     // Stores the interface/source from which cmd is sent.

  //Generate Random Permutation of N integers
  constraint c_random_permutation {
    G_order.size() == I_cmd_type.size;
    foreach(G_order[i]) {
      G_order[i] >= 0;
      G_order[i] < I_cmd_type.size;
    }
    unique {G_order};
    G_order[0] == 0;                             // Setting GO[0] to 0 for DUMMY INIT.
    G_order[G_order.size()-1] == G_order.size()-1;  // Setting GO[last] to last for spaced read.
  };

  //Create the matrix P by permuting the rows of I based on random G_order
  constraint c_matrix_P {
    foreach (G_order[i]){
      P_cmd_type[i]    == I_cmd_type[G_order[i]];
      P_be[i]          == I_be[G_order[i]];
      P_data[i]        == I_data[G_order[i]];
    }
    P_cmd_type.size() == I_cmd_type.size;
    P_be.size()       == I_be.size();
    P_data.size()     == I_data.size();
  };

  constraint c_matrix_M {
    M.size() == I_cmd_type.size;
  };

  constraint c_read {
    foreach (M[i,j]){
      //Equate data from reads in GO into memory model, if corresponding byte enables are set.
      (P_be[i][j] == 1) && (P_cmd_type[i] == READ)  -> (M[i][j] == P_data[i][j]);
    }
  };

  constraint c_write {
    foreach (M[i,j]){
      //Equate data from writes in GO into memory model, if corresponding byte enables are set.
      (P_be[i][j] == 1) && (P_cmd_type[i] == WRITE) -> (M[i][j] == P_data[i][j]);
    }
  };

  constraint c_atomics{
    foreach (M[i,j]){
      (P_be[i][j] == 1) && i>0 && (P_cmd_type[i] == ATOMIC_AND)     -> (M[i][j] == (P_data[i][j] & M[i-1][j]));
    }
  };

  //Conditions under which M rows do not change - reads or, writes and atomics with be=0
  constraint c_invariance{
    foreach (M[i,j]){
      if(i>0 && (P_cmd_type[i]==READ || (P_cmd_type[i] inside {WRITE, ATOMIC_AND} && P_be[i][j] == 0))){
        M[i][j] == M[i-1][j];
      }
    }
  };

  //This demo assumes a system where reodering is not possible from same source - Ordering constraints like this can modified by units
  constraint c_ordering_constraints {
    foreach (G_order[i]){
      foreach (G_order[j]){
        if(i!=j && i>j && SRC[G_order[i]] == SRC[G_order[j]]) {
          G_order[i] > G_order[j];
        }
      }
    }
   };

endclass
```

Figure 7. BATS simplified solver demo (Full Code: https://www.edaplayground.com/x/rXKN)

obtained in *Step 1* by drawing directed edges between MemOps of the same batch. ***Step 3***: Each edge obtained from *step 2* acts as a constraint to the solver for solving GO. We use the following rules to populate the sets S_{before} and S_{after} in *Step 1*. ***Rule 1***: Writes issued prior to the membar from the same source must belong to S_{before}. ***Rule 2***: Reads issued after the membar-ack from same or different source must belong to S_{after}. ***Rule 3***: Writes issued after the membar-ack (same or different sources) must belong to S_{after}. ***Rule 4***: Reads that received responses before membar issue (same or different clients than the membar) must belong to S_{before}. Please note that these set of rules are not comprehensive, and the rules might need modification based implementational details of membar in the design. However, by and large the method described above will still be applicable.

Let's now check how this will translate to the simple code-snippet shown in Fig. 8. For Membar issued by P0, Wr(A)D0 belong to S_{before} by Rule 1. Wr(F)1 is driven after membar-ack comes back to P0. Rd(A) is issued after Wr(F)1 is visible to P1. Combining previous two statements, we can see Rd(A) is issued after membar-ack returns to P0. Hence by Rule 2, Rd(A) belongs to S_{after}. Removing membar as per step2, leads to the constraint "Rd(A) should be ordered after Wr(A)D0". This is the exact constraint imposed by the membar in a producer-consumer code snippet. Although this might seem a bit complex, but it is easy to implement. Step 1 and 2 can be easily done using data collected by scoreboards and some graph processing algorithm. All it needs is information about drive time, membar-ack time, MemOp source etc which are observable on the DUT boundary. The question is in Step3 how do we write constraints which can change dynamically at runtime based on information obtained from Step 1 and Step2? This can be easily done by right padding matrix I with a square matrix V of size (B_s x B_s) . I_{new} = [I_{old} V]. This means every MemOp now has a bit-vector of size B_s which denotes which other MemOps should be ordered before or after the MemOp. The encoding is *V[i][j]==1* => MemOp in ROW_i should be ordered before MemOp in ROW_j. The matrix V can be populated using Step1 and Step2. We can then write a simple constraint on the ordering based on the values present in V. We provide a simple demo of this implementation in a separate EDA playground link: https://www.edaplayground.com/x/DsUc

VIII. RESULTS AND CONCLUSION

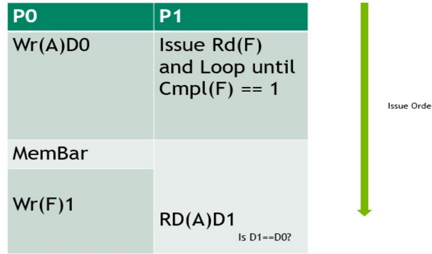
Table 1 shows a comparison of BATS to POSS and TSO tool on various aspects. Table 2 shows 10x reduction in engineering effort to deploy BATS over POSS scheme. Maintenance effort reduction for BATS over POSS is expected to even greater. This saving of engineering effort comes from the fact that BATS scheme is totally independent of micro-architecture. BATS does not probe any RTL signal to determine the GO. As such, any architectural changes to cache-coherency protocol, cache hierarchy or serialization logic does not affect the BATS checking methodology. However, this savings come at the cost of slight loss of temporal coverage due to stimulus-batching which is needed in order for SV solver to solve the constraints in a time-bound manner. BATS provides better coverage over TSO tool w.r.t atomics in a random simulation environment. We compared the mean wall-clock time and simulation cycles for BATS over POSS scheme. BATS showed a modest 10-15% reduction in simulation performance incurred due SV constraint solver overhead. However, it provides excellent portability across TBs due to the ability to specify ordering rules as high-level SV constraints and substantial reduction in development and maintenance cost. Therefore, we expect BATS to play a significant role in MOP verification along with TSO tool and POSS approach.

Table 1. Comparison of various existing approaches to MOP to BatchSolve

Approaches	Complexity	Dev-Cost	Robust	Scalable	Portability	Flexibility	PostSi?
TSO/Covert Tool	Polynomial	Medium	High	Yes	High	Low	Yes
Alloy/SAT Solver	Exponential	Low	High	No	High	High	Yes
POS	Linear	High	Low	Yes	Low	Low	No
BatchSolve	Linear	Low	High	Yes	High	High	No

Table 2. Comparison of Effort Estimates for POS vs BATS

	POS	**BATS**
Init Develop effort	80 weeks	8-weeks
Maintenance effort per project(estimate)	80 weeks	0-1 weeks (not including debug)
Porting effort to other UVM TB	Not portable easily	1 week (assuming TB has some score boarding)



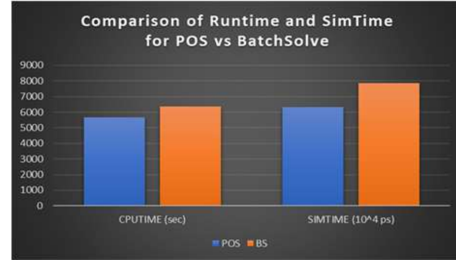


Figure 8. Producer consumer data transfer

Figure 9. Comparison of POSS vs BATS for Runtime/Simtime

Algorithm 1. 5 BATS Stimulus Batching API (Pseudo-code)

```
Input: Op (Random  MemOp generated by sequence)
Output: PASS/FAIL/COND_PASS
Data Structures: 2 Associative Arrays or Hashes: int OpCount[bit[63:0]]; time LastTime[bit[63:0]]
Pseudo-Code:
CacheLine_A = Op.Addr>>log_base_2(CACHE_LINE_SIZE);
If(!OpCount.exists[CacheLine_A]){OpCount[CacheLine_A] = 1; LastTime[CacheLine_A]=$time; return PASS}
Time diff_time = $time - LastTime[CacheLine_A];
FULL = (OpCount[CacheLine_A]==BATCH_SIZE)? 1 : 0;
ALMOST_FULL = (OpCount[CacheLine_A]==BATCH_SIZE-1)? 1 : 0;
SPACED = (diff_time > TIME_THRESHOLD) ?  1: 0;
if(FULL && SPACED) { OpCount[CacheLine_A] = 1; LastTime[CacheLine_A]=$time; return PASS}
if(ALMOST_FULL && SPACED) { OpCount[CacheLine_A] ++; LastTime[CacheLine_A]=$time; return
COND_PASS}
if((FULL || ALMOST_FULL)  && !SPACED){return FAIL}
OpCount[CacheLine_A] ++; LastTime[CacheLine_A]=$time; return PASS
```

Algorithm 2. Pseudo-Code for identifying spaced-reads in a batch

```
Input: op child_q[$], bit[63:0] sector_address;
Output: Void. Mark child_q[i].spaced_rd=1 for all spaced reads children child_q[i]
Data Structures: int sector_idx_q[$]; int cacheline_idx_q[$];
Pseudo-code:
child_q.sort with (item.get_SRC_drive_time());
Cacheline_address=get_cacheline_addr(sector_address);
sector_idx_q = child_q.find_index  with ((item.get_sector_addr() == sector_address));
cacheline_idx_q = child_q.find_index  with ((item.get_cacheline_addr() == cacheline_address));
Prune/Delete entries in cacheline_idx_q such that:
    a.  No two child with indices cacheline_idx_q have the same parent
    b.  To achieve above cannot prune any child which has address=sector_address
Iterate over the pruned cacheline_idx_q sequentially and mark every BATCH_SIZE child as a spaced read.
```

REFERENCES

[1] Cantin, Jason F., Mikko H. Lipasti, and James E. Smith. "The complexity of verifying memory coherence and consistency." *IEEE Transactions on Parallel and Distributed Systems* 16, no. 7 (2005): 663-671.

[2] Darbari, Ashish, et al. "Formal Modelling, Testing and Verification of HSA Memory Models using Event-B." *arXiv preprint arXiv:1605.04744* (2016).

[3] Alglave, Jade, Luc Maranget, and Michael Tautschnig. "Herding cats: Modelling, simulation, testing, and data mining for weak memory." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.2 (2014): 1-74.

[4] Chen, Yunji, et al. "Fast complete memory consistency verification." *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009.

[5] Hu, Weiwu, et al. "Linear time memory consistency verification." *IEEE Transactions on Computers* 61.4 (2011): 502-516.

[6] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, Juin-Yeu Joseph Lu and Sridhar Narayanan. “TSOtool: A program for verifying memory systems using the memory consistency model. ISCA '04: *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[7] C. Manovit and S. Hangal, "Completely verifying memory consistency of test program executions," *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, 2006, pp. 166-175, doi: 10.1109/HPCA.2006.1598123.

[8] https://xdevs.com/doc/Standards/PCI/PCI_Express_Base_4.0_Rev0.3_February19-2014.pdf