

# Mechanism to Generate FIFO VC Dependency Graph and Its Application to System Level Deadlock Verification

Debarshi Chatterjee, Chad Parsons, Siddhanth Dhodhi  
Nvidia Corporation. {debarshic, cparsons, sdhodhi}@nvidia.com

*Abstract*-System-level deadlock bugs are extremely elusive and can often escape pre-silicon verification and even emulation testing. If deadlocks are detected on silicon, it can delay time-to-market for the product. In the absence of a workarounds (through software patch, defeaturing in hardware etc.), a possible re-spin could cost the chip manufacturer millions of dollars [1]. Deadlocks at unit level can be found using formal verification techniques that exhaustively search all possible hardware states and legal input combinations. Applying formal verification to identify system level deadlocks in Register Transfer Level (RTL) implementation of modern large-scale design is challenging due to exponential complexity associated with exploring all possible design states [2]. Directed random testing though effective, require prior intuition and design knowledge of where the deadlock loops could be present. In most cases, those intuitions come from bug escapes in prior projects. In this paper, we present a method for finding deadlock loops in the design using a framework that learns dependencies across the various Virtual Channels (VCs) within First-In First-Out (FIFO) components and captures them in the form of a FIFO VC Dependency Graph (FDG). We applied our technique on unit and super-unit design verification environments of a state-of-the-art GPU design. It uncovered several interesting deadlock scenarios which were not apparent without this tool.

## I. INTRODUCTION

A hardware system deadlocks if it enters a state from which it cannot make any forward progress. This typically happens when one component  $A$  in a system is waiting for another component  $B$  to complete a certain task before it can make forward progress. If the other component  $B$  simultaneously enters a state in which it is waiting for  $A$  to complete another task before it can make forward progress on the task for which  $A$  is waiting for, then both are waiting for each other forming a circular dependency without each making any progress.

To explain this better, let us refer to a practical example Fig. 1. *Unit A* receives messages ( $Msg$ ) from the memory subsystem using the physical interface  $IF2$ , stores them in an internal FIFO, does some processing and then drives them off as Writes ( $Wr$ ) to the memory subsystem on  $IF1$ . These  $Wr$  driven to memory subsystem are processed by MMU for address translation and then sent to L2-cache and main memory. MMU simultaneously might receive barrier instruction from some other unit attached to the memory subsystem (not shown in figure). As MMU processes the barrier instruction, it will send  $Flush$  command on the interface  $IF2$  and wait for the  $FlushAck$  to be driven by *Unit A* on  $IF3$ . In the current implementation, let us assume  $Msg$  and  $Flush$  both flow on  $IF2$  with possible Head-Of-Line (HOL) blocking. If *Unit A* cannot drive  $Wr$  to  $IF1$  and it keeps receiving messages from  $IF2$ , eventually it's internal FIFO will become full and it will backpressure  $IF2$ . So, the dependency of stalling  $IF1$  will stall  $IF2$  is known (denoted by E1 in the figure). The memory subsystem needs to guarantee that if we stall  $IF2$ , under no circumstance will forward progress of  $Wr$  on  $IF1$  be hindered. This might need special consideration because stalling  $IF2$  could stall  $Msg$  as well as  $Flush$ . If MMU is processing a barrier instruction and a  $Flush$  is pending, it needs to make sure it can simultaneously handle address translations for  $Wr$  coming on  $IF1$ . If it cannot guarantee that, then it is likely that when *unit A* FIFO is full, it will back-pressure  $IF2$ , which will cause  $Flush$  to be pending in MMU, which in turn would back-pressure  $IF1$   $Wr$  drainage – eventually leading to a deadlock.

## II. PREVIOUS WORK

In order to keep hardware design deadlock free, concurrent protocols (such as cache-coherency protocols) that a hardware implement can be vetted for forward progress by deploying formal verification techniques [3][4] on the high-level abstraction of those protocols. Designers then try to keep the RTL implementation as close to the high-level protocol intent as possible. However, due to various reasons such as schedule pressure of implementing a

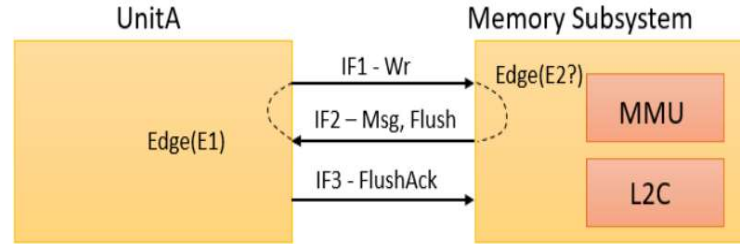


Fig. 1. Deadlock – A Practical Example

protocol within the scope of a legacy design, performance optimization etc., designers often diverge slightly from the high-level specification. It then becomes imperative to verify the deadlock free behavior of the design.

Darbari et al. [5] demonstrated an application of formal verification methodology to a design by tracking control signals and creating a set of assertions which detects whether the design has entered a deadlock state. They use formal and/or simulation-based environment to figure out whether those assertions are triggered to detect deadlock cases. For the formal tool to perform liveness check and detect deadlock on a block level design, constraints must specify all legal input combinations and timing considerations between the inputs and outputs of a design. In many cases, such a specification is not readily available for all blocks in a complex industrial design. Creating such specification is laborious. Moreover, scaling this method to system-level seems challenging because of complexity.

Several interesting applications of formal verification to system level deadlock detection were presented by M. Munishwar et al. [6] and J. Verdonck et al. [7]. Both papers rely on the fundamental approach to a design flow where an Architectural Model (AM) needs to be created for every block that is part of the system. The AM is lightweight and only includes aspects that ensure forward progress. The block level AMs are stitched together to form a system level AM. The system level AM is then verified using formal techniques to ensure there are no deadlocks. The block level AMs are implemented with constraints, which are typically verified in formal against the RTL implementation of the blocks. This approach has been demonstrated to work very well in many cases. However, for complex blocks in an industrial design, creating suitable and accurate AMs for all blocks in a system may require considerable engineering effort.

At Nvidia we use a mix of simulation-based deadlock verification techniques at the system level. These are briefly mentioned below: 1) *Random Stalling*: Randomly stall various unit interfaces and VCs either by introducing artificial flow control modules within the design or through artificially holding back transactions or credits using verification stubs. This technique is good at finding low hanging deadlock bugs. 2) *Independence checks on VCs*: Artificially stall a VC and make sure traffic over independent VCs can drain. These checks make sure the design does not violate the high-level architectural specification. When a particular egress VC is stalled, the specification provides expectations on which ingress VCs are expected to always stall, which ingress VCs are expected to always flow, and which ingress VCs could possibly stall. It is challenging to find such elaborate specification for all units in complex industrial designs. Even when such specification is available for a particular unit, it becomes difficult to conclude whether a violation of such specification will lead to a system level deadlock without knowing such specifications from all other units. 3) *Directed Random Testing*: When a possible deadlock loop is suspected, develop a directed deadlock verification test. Such tests usually stall a specific interface and drive directed traffic to the stalled interface. Once the backpressure from stalled interface is propagated to a desired interface, the test then drives and waits for independent traffic to drain out before releasing the artificial stall. We found this technique to be the most effective in finding corner case system level deadlock bugs. However, it suffers from the draw-back that it requires designers to narrow down on possible deadlock cases, which then can be verified through directed-random testing.

The open question then remains: How to automate the process of finding dependencies in the design implementation without any design knowledge? How to automate the process of narrowing down on possible deadlock scenarios without relying on human intuition? How to guarantee that, as the dependencies change over the course of the

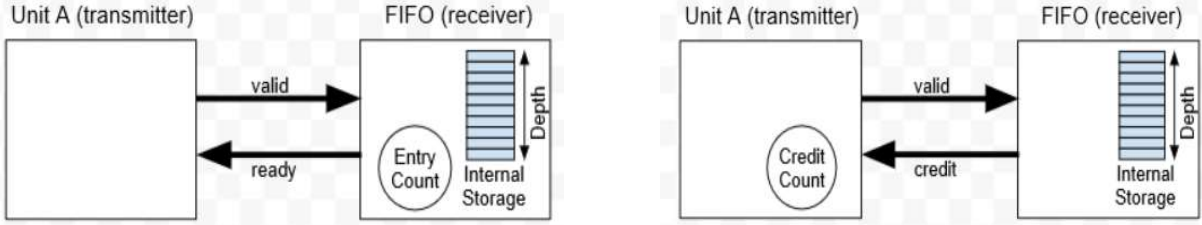


Fig.2 (a) Valid-Ready interface between unit logic and FIFO (b) Valid-Credit interface between unit logic and FIFO

project, due to implementation changes, we can still have confidence that the final design is deadlock free. In this paper, we demonstrate a data-driven graph-based approach to address that question.

### III. BACKGROUND

FIFOs are storage elements that are widely used in hardware design. In order to prevent overflow of the FIFOs, there are two types of interface control flows that are typically used: Valid-Ready and Credit Based. The former employs a handshake where the transmitter asserts a *valid* signal to initiate a transaction. The receiving FIFO indicates that there is space in the internal storage by asserting the *ready* signal any time the entry count is less than the depth of the internal storage. A transaction is completed when both the *valid* and *ready* signals are asserted in the same cycle. (see Fig. 2(a)). The latter uses a credit system where the transmitter maintains a credit count of the receiver's storage. When the transmitter has non-zero credits count and it wants to transmit a packet, it asserts *valid* and decrements its credit counter. When the receiving FIFO pops its storage, it indicates by asserting a credit signal back to the transmitter. At this point the transmitter increments its credit counter (see Fig.2(b)). In either case, the receiver can control the flow of traffic by de-asserting the *ready* signal, or no longer issuing credits on the *credit* signal when the internal storage has been fully populated (FIFO full condition). We define such a case as the receiver asserting "backpressure" on the transmitter. In many systems, FIFO depth is designed such that there will very rarely be backpressure to maximize performance and minimize stalls in functional pipelines.

Basic FIFO functionality is almost the same no matter where in the design they are used. A vast majority of FIFOs in Nvidia's design are generated using FIFO Generation script (FIFOGEN). Designers would customize certain FIFO attributes such as depth, interface protocols on input and output sides, credit classes (for threaded FIFOs), clock domains (for asynchronous FIFOs) etc. In order to generate the FIFO module, designers call the FIFOGEN with the required attributes. The generated FIFO module is then instantiated. Since FIFOs are ubiquitous in hardware design and have built-in flow control mechanism, it becomes a natural choice for artificially injecting stall and observing dependencies necessary for creating the FDG. We enhanced the FIFOGEN script to insert custom non-synthesizable RTL code within each FIFO generated by FIFOGEN. This will be explained in more details in the next section. It is worth mentioning here, that although we used FIFOGEN script to create the stalling framework, this work is extendable to design flows which do not have such FIFO generator scripts. The additional work in that case would be write a script that augments all FIFO module code with some custom code.

### IV. STALLING FRAMEWORK

The stalling infrastructure comprises custom non-synthesizable RTL code inserted within each FIFO module by the FIFOGEN script. The stalling implementation differs based on the protocol used on the write side. For valid-ready protocol on the write side, there is only one *Pause* signal inserted into the FIFO. When the *Pause* signal is forced high from the testbench (using SystemVerilog force commands), the FIFO would accept the first write request and then de-assert the *ready*. The *ready* signal will remain de-asserted for the entire duration of time for which *Pause* is held high. For valid-credit FIFOs with multiple credit classes, we add  $n$  separate *Pause* signals:  $PauseVC_1, PauseVC_2, \dots, PauseVC_n$ , where  $n$  denotes the number of credit classes in that FIFO. When  $PauseVC_n$  is forced high, the FIFO would hold sending back credits for credit class  $n$ . Clearly, for receiving FIFOs that are credit based, the source FIFO would not witness the stall until it has exhausted all the credits it had for that class.

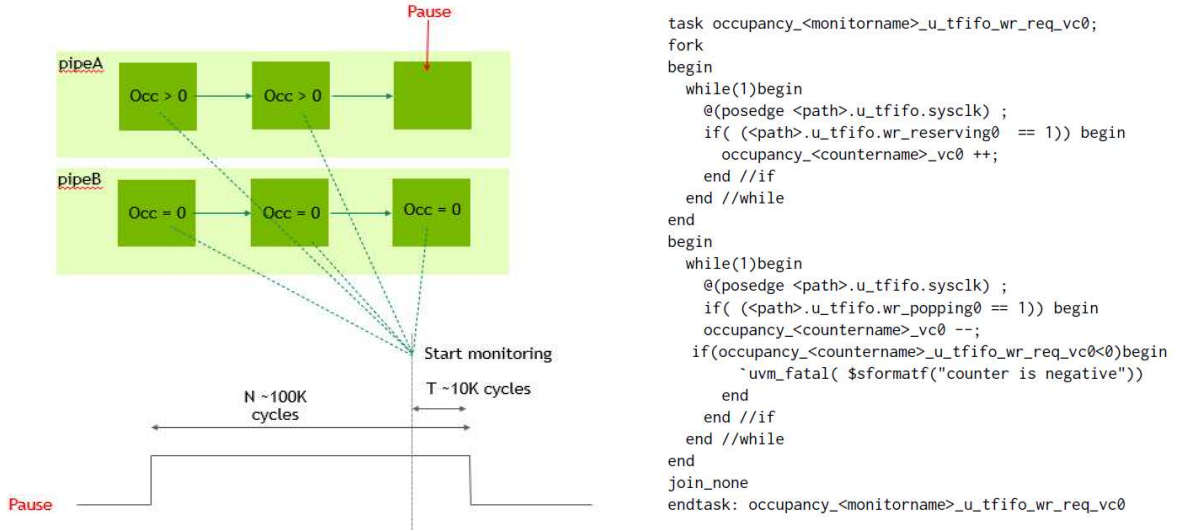


Fig. 3(a) Stall Control and monitoring sequence (b) Sample SV monitor code to measure per-VC occupancy

## V. STALL CONTROL AND MONITORING SEQUENCE

The stall control and monitoring sequence is a 3-step process as shown in Fig. 3(a). *Step1*: In a random test, randomly pick one FIFO  $X$  in the design on which the stall will be applied. *Step2*: Start stalling FIFO  $X$  i.e., assert *Pause* after a random time from reset de-assertion using the stalling framework described in *Section IV*. Hold the stall for a long duration ( $N \sim 100K$  cycles) while random traffic is flowing through the design. *Step3*: After  $N-T$  stall cycles ( $T \sim 10K$  cycles), start monitoring all FIFOs in the design and then compute the Dependent FIFO List (DFL) w.r.t stalling FIFO  $X$ , denoted as  $L_t(X)$ . Define  $L_t(X)$  for the test  $t$ , to be the unordered list of FIFOs (or FIFO VCs for multi-threaded FIFOs) that have non-decreasing non-zero occupancy from  $N-T$  to  $N$  stall cycles, when FIFO  $X$  is stalled. If the FIFO selected in Step1 is a multithreaded FIFO then  $X$  refers to a VC in the FIFO which can be stalled in Step2 by asserting *PauseVC<sub>n</sub>*. It can be proved that for large enough  $N$  and  $T$ , FIFOs in  $L_t(X)$  are dependent on  $X$ .

## VI. CONSTRUCTING THE FDG

We run millions of directed-random simulations. Let us suppose, in  $k_i$  such tests, FIFO (or FIFO VC)  $X_i$  was stalled and  $L_0(X_i), L_1(X_i), \dots, L_{k(i-1)}(X_i)$  denotes the DFL for each of these  $k_i$  tests respectively. Let's define Unified DFL  $L(X_i) = L_0(X_i) \cup L_1(X_i) \cup \dots \cup L_{k(i-1)}(X_i)$ . The FDG  $G = (V, E)$  is defined as a set of vertices  $V = \{X_i | i=1, 2, \dots, n\}$  and a set of directed edges  $E = \{(X_i, X_j) | X_i, X_j \in V \text{ and } X_j \in L(X_i)\}$ , where  $L(X_i)$  represents Unified DFL. Next, DL loops are detected by running cyclicity checks on the FDG using Algorithm shown in Fig. 5. Entire flow is shown in Fig. 6

## VII. PROPERTIES OF THE FDG

*Existence of edge guarantees dependency between the FIFO VCs but the absence of an edge does not guarantee independence*: The existence of an edge in FDG guarantees at-least one test condition under which the dependency exists. For every edge in the graph, it is important to store the Changelist-Testname-Seed (CTS) combination that generated that edge. Once a loop is detected, the CTS information on the edge can be used to reproduce the simulation which revealed the unexpected edge in the graph. The absence of an edge between two nodes in the FDG, however, does not guarantee independences of the two nodes. This is true because, we can never be sure when we are done discovering all the edges in the FDG. It is possible that the random simulation did not have the specific test stimulus which would expose an edge in the graph. However, random regression test-suite goes through rigorous coverage analysis to make sure it satisfies goals for line coverage, toggle coverage, expression coverage and functional coverage. Hence, we expect, the FDG to reveal even corner case dependencies across all FIFO VCs.

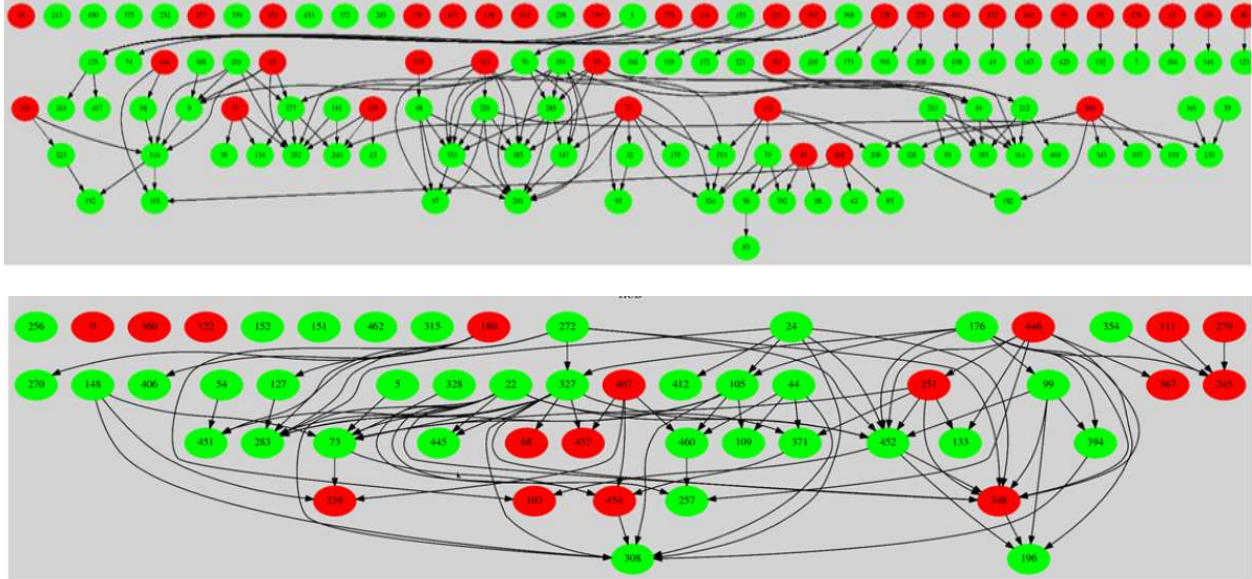


Fig.4. FDG generated for 2 modules (red are valid-ready FIFOs and those marked in green are credit classes for valid-credit FIFOs)

**Algorithm 1:** Remove Nodes to make Graph  $G$  acyclic and print out cycles involving deleted nodes

**Data:** Directed Graph  $G = (V, E)$  which could be cyclic or acyclic

**Result:**  $G' = (V', E')$  where  $G'$  is acyclic and  $V = V' \cup X$  where  $X$  is the set of excluded FIFOs to make  $G'$  acyclic

```

1 initialization;
2  $G' \leftarrow G$ ;
3  $X \leftarrow \emptyset$ ;
4  $Done \leftarrow 0$ ;
5 do
6    $Done \leftarrow \text{ClassifyFifoLevels}(G')$ ;
7   if  $\neg Done$  then
8      $x \leftarrow \text{SelectNodeToRemove}(G')$  where  $x \in V'$ ;
9      $X = X \cup \{x\}$ ;
10    DetectCycle( $x$ );
11     $V' = V' - \{x\}$ ;
12    for  $e' \in E'$  do
13      if  $\exists w \in V' \mid e' = (x, w)$  or  $e' = (w, x)$  then
14         $E' = E' - \{e'\}$ ;
15 while  $\neg Done$ ;

```

**Algorithm 2:** Procedure ClassifyFifoLevels

**Data:** Directed Graph  $G = (V, E)$  which could be cyclic or acyclic

**Result:** TRUE if  $G$  is acyclic, FALSE if  $G$  is cyclic  
If  $G$  is acyclic, gen mapping function  $\zeta$  which maps every vertex to a non-negative integer,  $\zeta : V \rightarrow \mathbb{Z}^+ = \{0\} \cup \mathbb{Z}^+$

```

1 initialization;
2  $Level \leftarrow 0$ ;
3  $N \leftarrow 0$ ;
4  $\forall v \in V, \zeta(v) \leftarrow -1$ ;
5 do
6    $LevelNCount \leftarrow 0$ ;
7   for  $v \in V$  do
8     if  $\zeta(v) < 0$  then
9       if  $\nexists w \in L(v) \mid \zeta(w) < 0$  then
10         $\zeta(v) \leftarrow Level$ ;
11         $LevelNCount ++$ ;
12         $N = N + LevelNCount$ ;
13    $Level ++$ ;
14 while  $LevelNCount \neq 0$ ;
15 if  $N = |V|$  then
16    $result \leftarrow 1$ ;
17 else
18    $result \leftarrow 0$ 

```

Fig. 5. Algorithms for classifying FIFOs into levels and detecting DL loops

TB	N	M	Loops/Bugs
TB-Sunit1	1068	3717	2
TB-Unit2	55	45	1
TB-Unit3	83	99	0

Table1. Showing results of FDG over several TBs (N=Num. of Stalling FIFOs; M=Num. of non-stall FIFOs)

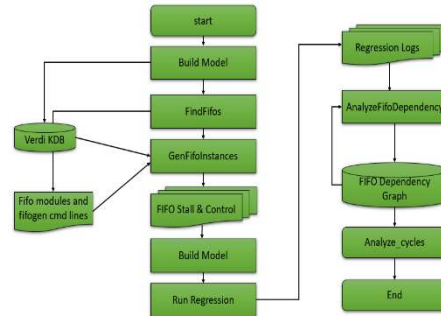


Fig.6. Flowchart for FDG generation

*FDG does not give temporal order of stalling within the list of stalled FIFOs:* For example, if  $X, Y, Z$  are three FIFOs in a linear-data-path, with data flowing from  $Z$  to  $Y$  to  $X$ , then the FDG will have edges  $X \rightarrow Y$  and  $X \rightarrow Z$ . This means stalling  $X$  stalls both  $Y$  and  $Z$ . In this case, stalling of  $Z$  does not happen unless  $Y$  gets stalled. However, FDG does not capture the temporal order of stalling.

*If a loop exists in FDG and the edges involved in the loop are transitive, then we have a possible deadlock scenario:* Strictly speaking we cannot guarantee that the edges in FDG are transitive. Hence a deadlock cannot be concluded from the existence of a loop in the FDG. In other words, a loop in the FDG is a necessary but not a sufficient condition for the existence of deadlock. For example, if FDG contains  $X \rightarrow Y$  and  $Y \rightarrow X$ , we cannot definitively conclude that there exists a test under which  $X \rightarrow Y$  and  $Y \rightarrow X$  co-exist. To prove this, consider the counter example, where the edges  $X \rightarrow Y$  is exposed only in a test where a specific Register Config = A and  $Y \rightarrow X$  is exposed only in test when the same Register Config = B, with these modes being mutually exclusive ( $A \neq B$ ). Then in this example, a loop is not sufficient to prove the existence of a deadlock. If the edges in the path of a loop are not transitive, we could get a "fake-loop" in FDG that does not lead to a deadlock. Designers however make sure that the design is free from cyclic dependencies and do not rely on non-overlapping modes to resolve cross-dependencies. This is consistent with the empirical data that we obtained by constructing FDG (with thousands of nodes and several thousand edges) on complex industrial design involving hundreds of modules. We found very few loops and no "fake-loops" caused by non-transitive property of the edges. Hence the approach we follow is that once we find a loop in the FDG, we flag that as a potential deadlock scenario and make RTL changes to eliminate that loop. By making the FDG acyclic we remove the necessary condition for deadlocks.

## VIII. SOME PRACTICAL CONSIDERATIONS

*TestBench(TB) infrastructure for stall control and monitoring:* In order to develop the TB infrastructure explained in Section V, one major practical consideration is: How to figure out the full hierarchical instance path for each FIFO in the design? Given a module-name, Verdi provides the search capability to identify all instances of that module in the design and extract the full hierarchical path to each instance. In the Verdi GUI, go to "Source→Find Signal/Instance/Instport", then select "Instance" from the drop-down menu and enter the module name. Verdi will find out all instances of the module. Verdi also provides a Command-Line-Interface (CLI) which can be called from within any script to achieve this task. Since all instances of a FIFO module follows the same naming convention for request/valid/credit/clock signals, once the hierarchical path to each instance is known, writing stall control and monitoring infrastructure in Section V is pretty much straight-forward. Since we create the stall control and monitoring related SV files dynamically using Verdi CLI search, the flow (explained in Fig. 6) is immune to design changes.

*Asynchronous FIFOs:* For FIFOs that have read and write pointers in different clock-domains, the SV monitor code (that calculates the per-VC occupancy) needs to make sure the increment-decrement operations are done based on signals synchronized to the same clock-domain.

*Choice of (N,T):* N Depends on the longest running test in the regression suite. The stall duration N should be large enough to guarantee all traffic in the test to drain in the absence of the artificially introduced stall. Since N is large,  $T=10\%N$  should be good for the purpose of FDG generation.

*No-Stall FIFOs:* Certain FIFOs are oversized by design. Special care must be taken to exclude these FIFOs from being stalled, since they can give rise to illegal edges in the graph. Stalling no-stall FIFOs result in illegal behavior. Hence random stalling regression often have a specification on which FIFOs are oversized and should not be stalled.

*Termination Criteria for Graph Generation:* Currently we stop generating the graph if we cannot find a new edge in the graph for several consecutive nightly regression runs. In future we plan to use coverage goals as the stopping criteria for generating the graph.

## VIII. RESULTS

We applied FDG DL Detection tool on a state-of-the-art GPU design across 3 testbenches (TBs) – One memory subsystem super-unit TB (TB -SUnit1), and 2 Unit level TBs (TB-Unit2 and TB-Unit3). The tool was run close to project tape-out when most DL verification testplan was executed. We debugged FDG loop with designers and found

several interesting cases. These are documented in bugs mentioned in Table 1. One such case, was a dependency loop found between *SystembarPendfifo (F1)* in UnitA and *CpuResponseFifo (F2)* in UnitB. Unit A has logic that is responsible for sending out membars, collecting the acks and returning the acks back to the *clients*. Membars are used by *clients* to make sure that writes are pushed to point-of-coherency by the time the ack is returned to the *client*. This is used for thread synchronization across *clients*. Membar.sys is a special type of membar. Acks for Membar.sys return to UnitA through IfB2A-Blocking VC. On stalling *F2*, systembar-acks were blocked. Hence, we found test where a systembar was sent out, *F2* was stalled, and the systembar-ack did not return to UnitA. This exposed the dependency edge between  $F2 \rightarrow F1$ . On the other hand, stalling *F1*, stalls any UnitA clients requesting a membar, including IfB2A-Blocking VC, and hence *F2*. We found a test, which artificially stalled *F1* and test stimulus drove CPU downstream reads on IfB2A-Blocking VC after the stall. Hence it exposed the edge  $F1 \rightarrow F2$ .

## IX. NOVELTY AND ADVANTAGES

An advantage of FDG based DL detection method is that it is completely automated and does not require any prior knowledge of the Design Under Test (DUT). To the best of our knowledge, there is no prior published work on the automated construction of a dependency graph across hardware components without using any designer specification. Since the FDG tool is built on top of FIFOGEN and uses Verdi search capabilities, it is capable of vertical and horizontal re-use (can be ported across TBs and projects with ease). We have ported this tool from super-unit TBs that uses UVM to non-UVM legacy unit level TBs. The tool also generates all the System-Verilog (SV) collateral on-the-fly. Hence there is no maintenance cost when FIFOs get added or removed from the design. The cost of developing this tool is ~30days of engineering effort (EE). However, porting to a new TB takes ~2 days of EE. There is no maintainability cost other than debugging the loops that are detected by the tool.

## X. LIMITATIONS AND FUTURE WORK

Although extremely useful, the FDG DL detection tool in its current form will not be able to identify deadlock loops outside the realm of FIFOs. Therefore, our future work is to identify other modules that are ubiquitous in the design and have built-in flow control in them. By introducing the stalling framework (described above) within such modules, we can include them in the dependency graph generation flow.

## XI. ACKNOWLEDGMENT

The authors would like to acknowledge the support of Roger Sabbagh and Ashish Darbari for discussions regarding their previous work on this topic.

## REFERENCES

- [1] T.R. Halfhill. The truth behind the Pentium Bug, March 1995; <http://www.byte.com/art/9503/sec13/art1.htm>.
- [2] Adnan Aziz, Vigyan Singhal, and Robert K Brayton. 1993. Verifying interacting finite state machines: Complexity issues. University of California at Berkeley (1993).
- [3] Patrice Godefroid and Didier Pirotin. 1993. Refining dependencies improves partial-order verification methods. (1993), 438–449.
- [4] A Nico Habermann. 1969. Prevention of system deadlocks. Commun. ACM 12, 7 (1969), 373–ff.
- [5] Ashish Darbari and Colin McKellar. 2017. Deadlock detection in hardware design using assertion based verification. (Sept. 19 2017). US Patent 9,767,236.
- [6] M. Munishwar, N. Zaman, A. Jain, H. Singh, V. Singhal, “Architectural Formal Verification of System-Level Deadlocks”, DVCon 2018.
- [7] J. Verdonck, K. Liatakis, K. Nsaibia, D. Gupta, S. Dewangan, T. Upadhyay, H. Singh, R. Sabbagh, “Covering the Last Mile in SoC-Level Deadlock Verification”, DVCon 2019.