A Modern Debug Paradigm: Python Visualization for DDR Memory Controller's Performance Analysis

Pallavi Kumar, AMD, San Jose
Michael Chan, AMD, San Jose

***Abstract*- For verification engineers, whether performance or functionality, traditional waveform viewers have been the go-to debug aid. However, as design complexity increases, these tools can become cumbersome and inefficient for deep analysis. Additionally, verification engineers often report that generating a waveform is time-consuming and requires significant disk space. This paper presents a novel approach to DDR memory controller's performance verification by leveraging Python-based data visualization as a primary debug aid. By extracting key performance metrics and signal behaviors from simulation outputs and rendering them through intuitive plots, faster and more insightful understanding of the test's behavior is achieved. This paper demonstrates how this approach complements or even replaces traditional waveforms in specific scenarios, offering a more scalable and customizable verification workflow. This approach is very relevant, more so for anyone working in performance verification, because root causing performance slacks are akin to finding a needle in a haystack, so the more data points gathered the better. This paper lists all the Python-based data visualizations that were added utilizing the existing log files printed by the universal verification methodology (UVM) monitors and demonstrates how it helps pinpoint problematic areas.**

I. INTRODUCTION

Modern systems on chips (SoCs) are designed by integrating several intellectual properties (IPs) using various interconnect layers. Although the exact functionality of the device is of the highest importance, the correct behavior in terms of performance is a crucial factor [1]. To gain competitive advantage, every device that gets taped out should be exhaustively tested to ensure that it meets all the required and defined performance metrics. Plus, optimizing the device's performance and minimizing the risk are very important aspects to placing the product on the market at the appropriate time [2].

Within SoCs, verifying that the memory controller and the network on chip are not the bottlenecks is of utmost importance. Many SoC performance verification teams often prioritize these blocks as the critical paths. Thus, it is imperative for the DDR memory controller team to run exhaustive performance testing across different DRAM memories (DDR*, LPDDR*), speed grades, address access patterns (linear, random), features (like inline-ECC, encryption), etc., and characterize the expected performance. These permutations result in an average of 7,000 test cases per controller IP. To add to the complexity, test results from register-transfer level (RTL) simulation runs should be compared against the results from the same test run on the architectural model and any differences in the performance metrics between the two models above or below a specified margin tolerance need to be debugged. Performance bugs are harder to debug than functional bugs, where exhaustive checkers and assertions provide clear clues.

Relying solely on the waveforms or manually parsing interface log files can take an average of two to three days to debug. By using post-processing scripts listed in this paper, parsing the log files generated by UVM monitors provides an excellent insight into the test's behavior, aiding in identifying issues with the test bench and pointing to the problem areas where performance degradation happened. The script can also parse the log file generated by the architectural model to generate similar visualizations that aid quicker debug convergence.

Python-assisted visualization can pinpoint the problem area within ten minutes, approximately a 99.3% reduction in debug time. AI-assisted coding tools were used to accelerate script development, enabling rapid prototyping and iteration, so the important thing was to come up with an idea. It is to be noted that any IP or SOC verification team, that is doing functional, or performance verification can benefit from the ideas presented in this paper to boost the debug efficiency and turnaround time.

II. DDR MEMORY CONTROLLER'S PERFORMANCE VERIFICATION METHODOLOGY

This section gives an overview of DDR memory controller's performance verification methodology. The memory controller design under test (DUT) supports different memory technologies, like double data rate (DDR) & low

power double data rate (LPDDR). The goal of performance verification is to ensure that the DDR scheduler can keep up with the demand bandwidth without compromising on the DRAM efficiency [3]. Fig. 1, illustrates the memory controller's performance verification methodology. There are two existing Python scripts, one of which, during the post simulation phase, reads the UVM monitor-generated log files at the DRAM interface and at the input ports to calculate statistics on latency, bandwidth, DRAM efficiency, average read/write switching etc. The other script then compares these statistics with the corresponding values from the architectural model's results database and declares the test as PASS or FAIL based on the predefined tolerance limit. Based on these pass/fail criteria, even if ten percent of tests out of these 7,000 fails, there are 700 test cases that need to be debugged. Thus, a debug aid that provides a quicker turnaround is a necessity.

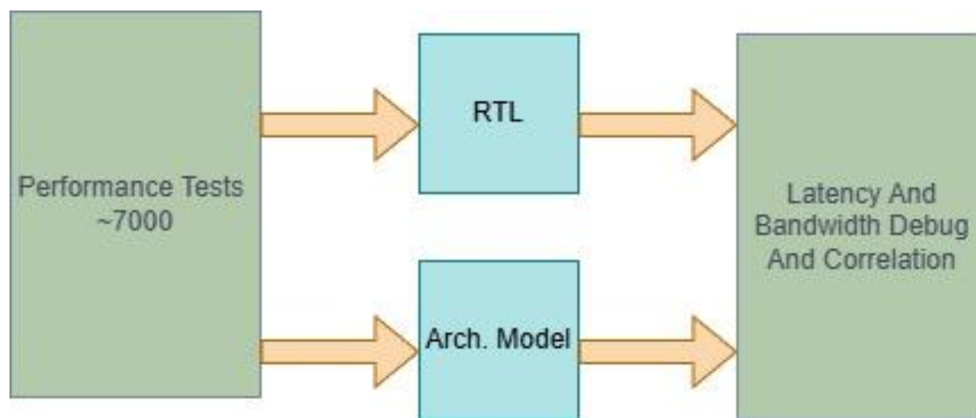

Figure 1 DDR Memory Controller's PV Methodology.

III. PYTHON ASSISTED DEBUG VISUALIZATION

This section gives an overview of the DDR memory controller's testbench set up, introduces the scripts that have been added at the input and output interfaces and finally presents some sample plots & debug cases.

Fig. 2, shows a simplified representation of DDR memory controller's test bench, showcasing the logs that are useful for performance verification. There are two network on chip input ports from where DRAM requests and responses are sent to/from and at the output is the DRAM which can be DDR* or LPDDR*. UVM monitors are in place to snoop into the transactions that are sent at both these interfaces. The transaction information gets logged into files called input_port_0.csv, input_port_1.csv, ddr_tracker.csv or lpddr_tracker.csv on the DRAM side. Input transaction log files give information on address, transaction type (read / write), timestamp and the tracker at the DRAM gives information on the type of DRAM command scheduled along with the row, bank, bank group, column information. The input tracker files help calculate the bandwidth achieved at the ports and the latency values, whereas the DDR/LPDDR tracker files are used in calculating the DRAM metrics.

The primary goal of the Python scripts is to provide an intuitive way to debug failures. With that in mind, listed in the below table are all the scripts along with their utility.

Script	Input Files	Intent	Utility
plot_moving_bandwidth.py	input_port_*.csv files	Plots the simple moving average (SMA) of the read and write bandwidth seen on both the input ports over a pre-defined interval. For reads, an SMA of average latency is also generated	It is useful in debugging failing test cases that report less than expected bandwidth; the time where a bandwidth drop is seen can be inferred by viewing this plot
calculate_bw_from_csv.py	input_port_*.csv files	Visualizes the bandwidth seen per read/write thread at input ports	It is useful in debugging read/write testcases where this plot can easily flag bandwidth imbalances among the threads
plot_per_bank_activity.py	ddr/lpddr tracker.csv files	Plots the distribution of critical DRAM commands per bank, bank group pair	It is particularly useful to debug a linear test case with bank group interleaving enabled wherein every thread is expected to open two/four/eight bank groups
dram_timings_analysis.py	ddr/lpddr tracker.csv files Arch. model's ddr tracker files	Creates a histogram of the critical DRAM timing parameters, one per timing parameter, for memory controller and the architectural model	Histogram plots between the two models give an insight into the range of values the schedulers had to pick for a timing parameter and how often

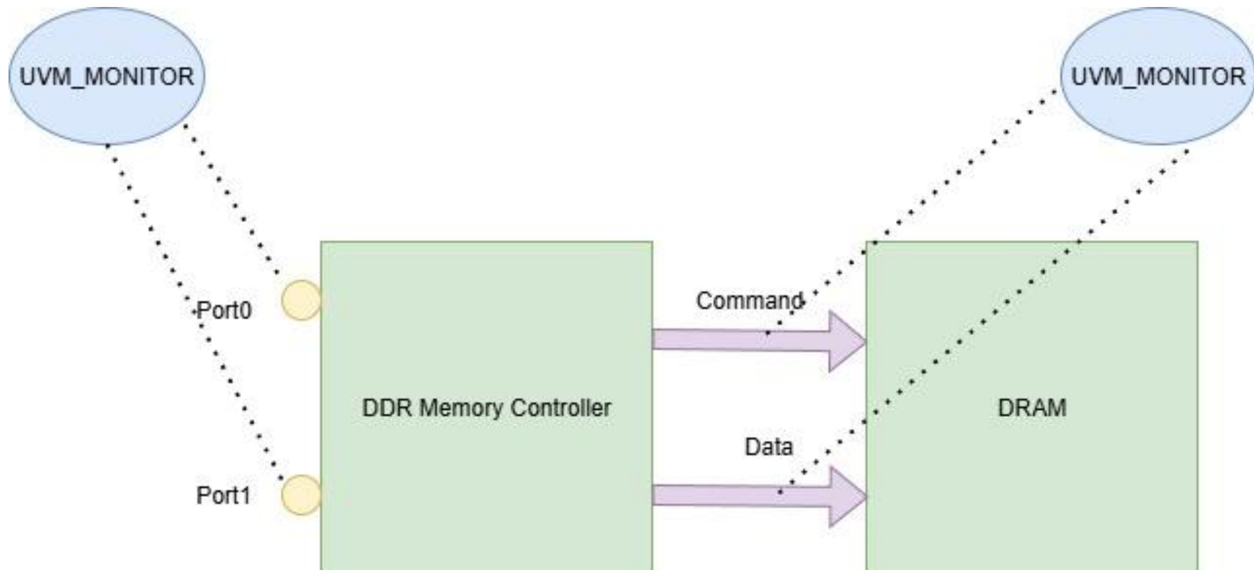

Figure 2 Simple Representation of DDR Memory Controller's Testbench.

All of these scripts are automatically called at the post simulation phase of a test, and the plots are saved as. jpegs in the test run directory. The memory usage of these plots is an average of 0.01 MB. It is obvious that on top of all the advantages these plots offer, memory usage is also low, making it an attractive option. The average memory occupancy of a traditional waveform database is approximately 2 GB to put things into perspective.

III. USE CASES AND DEBUG EXAMPLES

A. Moving Bandwidth SMA

Fig. 3, shows the SMA of a read master doing a linear access pattern which is the most efficient with a lot of page hits. As expected, the bandwidth (with the scale on the LHS axis) seen at the input ports is consistent and latency values (with the scale on the RHS axis) are also smooth. Fig. 4, on the other hand, shows a read master doing random access patterns on the same port as Fig. 3, which can have a mix of page hits and page misses. If there is poor performance by the controller, it is easy to find out the problematic time and skew the debug in that direction using this plot. This plot gives us a clue on the timestamp where the bandwidth drops, which is then used as a debug starting point when generating the waveform. Other plots that are described in this paper are also reviewed before generating the waveform to get as much debug information as possible.

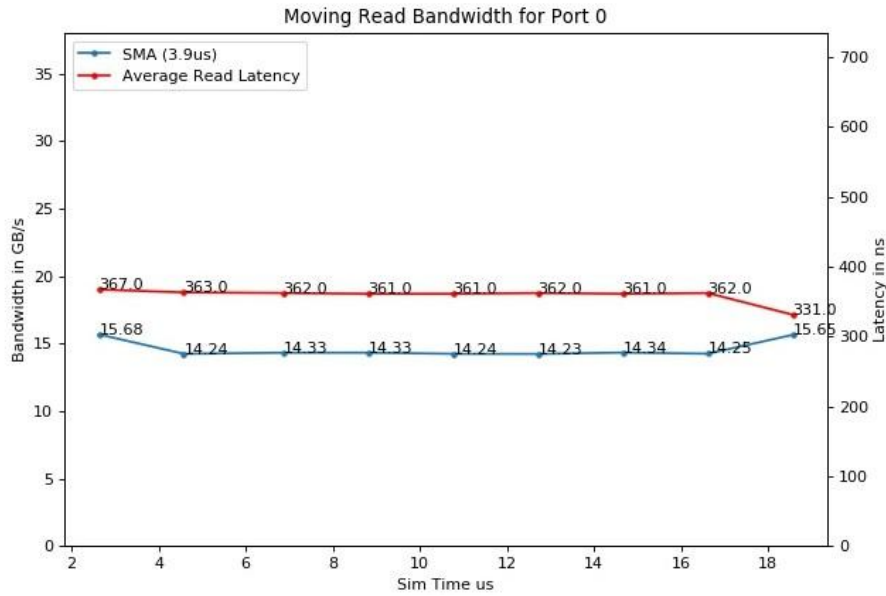

Figure 3 Example Plot of a Read Bandwidth SMA In Port 0 For A Linear Test.

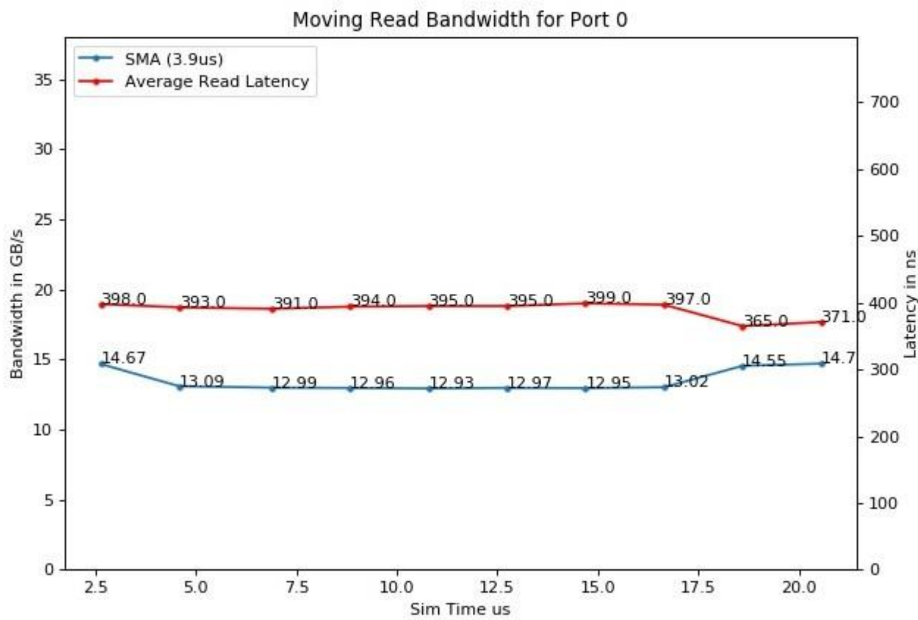

Figure 4 Example Plot of Read Bandwidth SMA in Port 0 For A Random Test.

B. Per Bank Activity Plot

The bank activity plot was developed to identify if a test failure is due to incorrect address generation or an incorrect testbench setup issue. If the bank activity plots are in line with the expectations, then the reason for a performance mismatch is most likely a design issue. Since linear address patterns are predictable and have bank group

interleaving enabled, each read or write thread is expected to open two/four/eight bank groups depending on the interleaving granularity. Fig. 5, is a snippet from a subsystem linear test that was running eight read masters to DDR with four bank group interleaving, so the expectation was that all 32 banks needed to be accessed. But this was not the case. The test was underperforming, and the issue was with the incorrect stimulus. This was debugged by simply viewing this plot, without having to parse through multiple log files or opening a waveform which at a subsystem can be very slow. Fig. 6, shows the correct and expected plot for this test post fix, the uniformity of Command Address Strobe (CAS), Precharge (PRE) and Activate (ACT) from the plots proves that the test and the scheduler are doing the right thing. X-axis in the plot shows the bank group / bank pair and the Y axis is the command count.

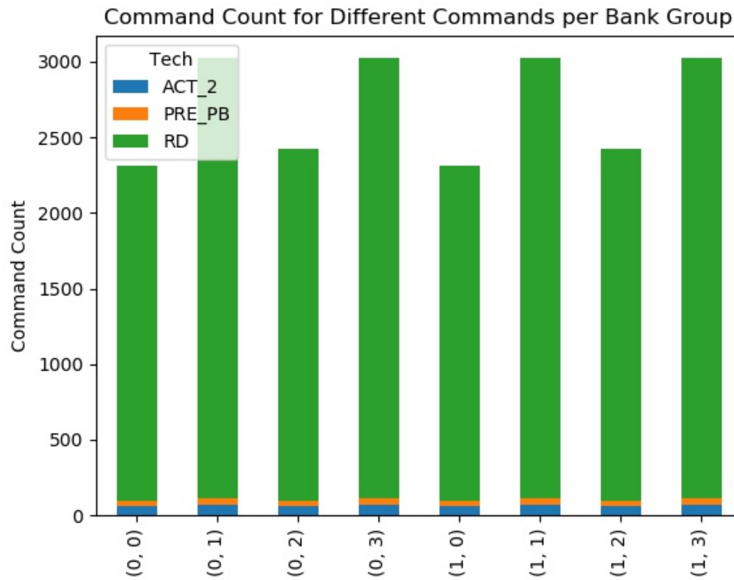

Figure 5 Plot Showing the Incorrect Bank Activity from A Sub-System Test.

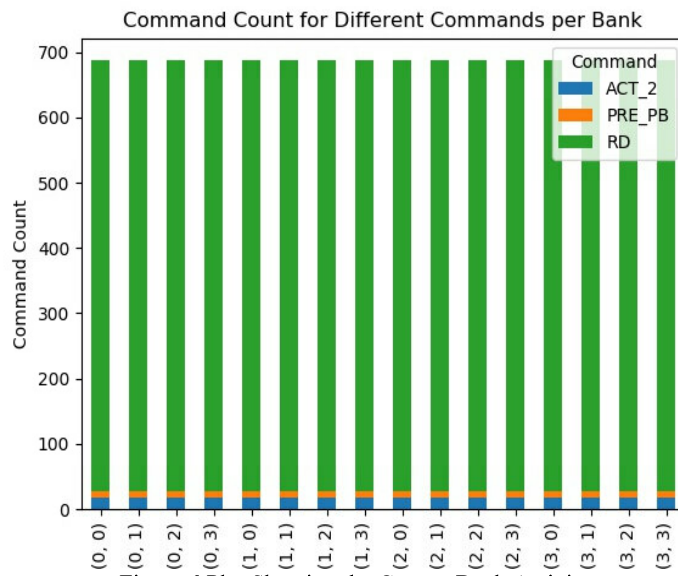

Figure 6 Plot Showing the Correct Bank Activity.

C.DRAM Safe Timing Analysis

Provides an insight into the actual timing parameters that were used in a run by the schedulers in RTL and architectural model and to give a comparison of the timing parameters used by both models. The script takes ddr/lpddr interface tracker files and architectural model's DRAM activity log as inputs and generates a histogram showing the range of values picked for every safe timing parameter along with their frequency. Shown here in Fig. 7, is a DDR linear write test with eight bank group interleaving. The script has generated a combined histogram of

column address strobe (CAS) to column address strobe (CAS) spacing used in this test and has characterized the values into four bins: CAS to CAS short(=tCCD_s), CAS to CAS long (tCCD_l), between the two and greater than tCCD_l. The CAS scheduling difference between both models is apparent from the plot. This gives a clear insight into the scheduling differences between the two models. Based on the analysis from this plot, the model that does not comply with the expected scheduling pattern is tweaked and the tests are re-run. Instead of generating a waveform and waiting to review it, here the tests are already rerun with a potential fix. Former process would take up to 4 days, while the latter takes a day.

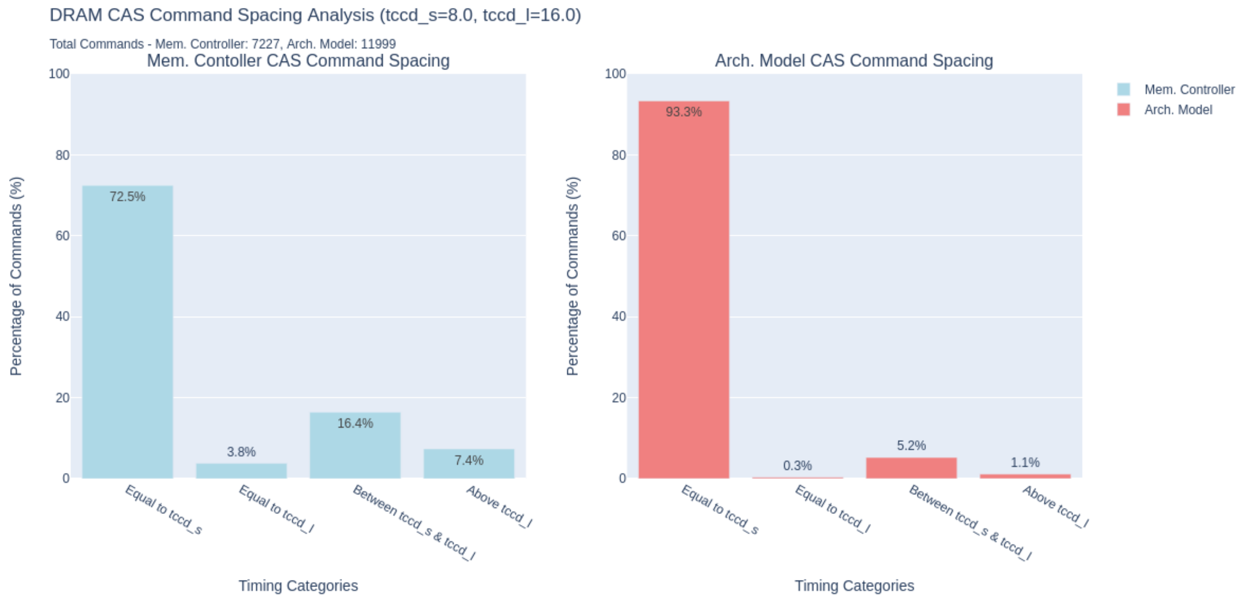

Figure 7 Command to Command Spacing Histogram Between RTL & Architectural Model.

D. Bandwidth Per Thread

This plot is very useful to debug a read/write ratio test, since it helps to visually spot check and confirm that no one thread is taking up the entire bandwidth. Fig. 8, shows an example of an eight read/write 50/50 linear test case run in LPDDR with two bank group interleaving. The test is expected to have some bank "fighting" since 16 threads times two equals 32 banks are expected to be accessed, which is two times the number of available banks in LPDDR (16). This implies that each bank has one read, and one write thread accessing it. Although bandwidth hogging is a possibility in this test, the plot clearly shows that the bandwidth is equally distributed among all masters.

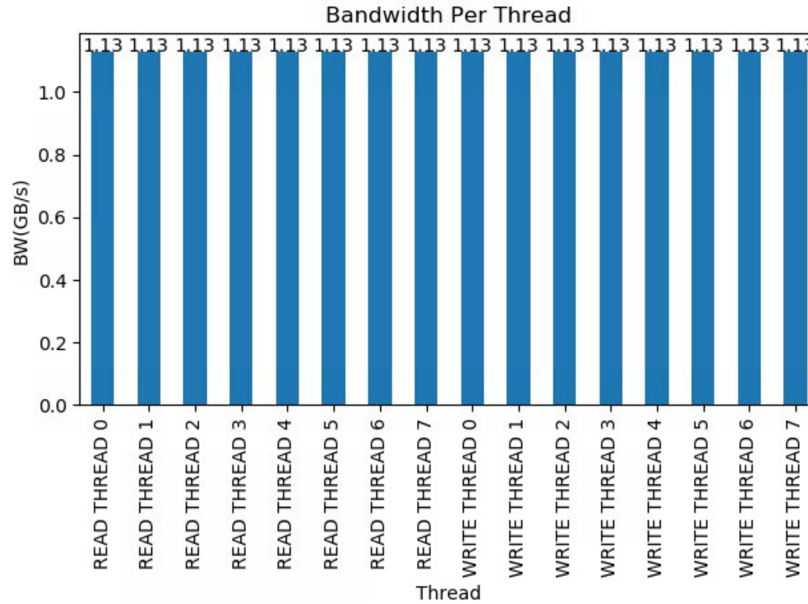

Figure 8 Example Plot of BW Per Thread.

IV. SUMMARY

This paper presents a novel idea of using Python's visualization techniques to advantage by using the logs generated by the UVM monitor. These plots are more intuitive for debugging than the traditional waveforms. Sample plots shown from each script demonstrate their debugging power. They also consume very little disk space as opposed to the memory-heavy FSDBs, thereby making them a very viable option to consider. These plots are expected to become the new go-to debug tools. But despite their advantages, these plots do not replace traditional waveforms. To root cause the actual design bug, a waveform is still needed. This methodology does not rely on external tools for visualization. Instead, it uses lightweight Python scripts to generate intuitive plots, making it easy to adapt and integrate. DDR Memory Controller team has already seen a huge productivity boost in debug time with the aid of these plots and is actively working on proliferating this methodology to other teams.

REFERENCES

[1] P. Ghosh and R. Srivastava, Case Study: SoC Performance Verification and Static Verification of RTL Parameters, in Proc. IEEE Int. Workshop on Microprocessor/SoC Test, Security & Verification (MTV), 2019, pp. 65–72. DOI: 10.1109/MTV48867.2019.00021
[2] P. Ghosh. D. Mai, A. Chopra and B. Sood "Self-Checking Performance Verification Methodology for Complex SOCs "
[3] DDR read reorder whitepaper by M. Greenberg, Synopsys