AMD

2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

A Modern Debug Paradigm Using Python Based Visualization

Pallavi Kumar, Michael Chan, Advanced Micro Devices Inc.

Agenda

- A Background On Performance Verification
- Traditional Waveform viewers and their drawbacks
- DDR Memory Controller’s PV methodology
- A Modern Debug Paradigm
- DDR Memory Controller’s Testbench
- List of Python Scripts And Their Utilities
- Example plots
- Example use case
- Conclusion

A Background On Performance Verification

- Although device functionality's verification is of utmost importance , verifying correct behavior in terms of performance is crucial
- 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.
- Within a SOC (system on chip) 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 RAS, encryption), etc., and characterize the expected performance
- Performance verification tests most often do not have a plethora of UVM checkers or assertions to pinpoint the exact root cause making it more difficult to debug than a test that is checking functionality

Traditional Waveform viewers and their drawbacks

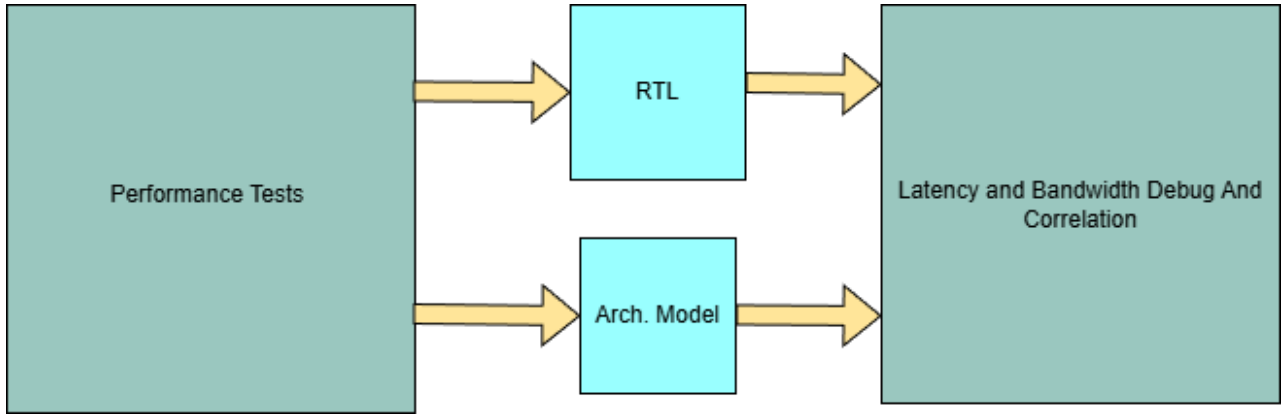
Traditional waveform viewers have always been the go-to debug aid for all verification engineers

However, as design complexity increases, these tools can become cumbersome and inefficient for deep analysis

Generating waveforms are time consuming plus they consume disk space

DDR Memory Controller's Performance Verification Methodology

- Tests are run in an architectural reference model and RTL
- There are over hundreds of test cases per device that cover different memory technologies (DDR*/ LPDDR*) and features (RAS/Refresh types, etc.)



- Since we are running our tests in 2 sets of models, we spend quite a lot of time in debugging and correlating between the 2 models on the bandwidth and latency failures seen in our regressions
- Relying on traditional waveforms and parsing log files for debugging will take formidably long
- Thus , an effective solution that can cut the debug turnaround time is a necessity

A Modern Debug Paradigm

Relying solely on the waveforms or manually parsing interface log files can take an average of two to three days to debug

Python based visualization of UVM monitor generated logs offer a more intuitive debug framework

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 presentation will touch upon the python-based visualizations that were added to aid performance verification debugs of DDR Memory Controller

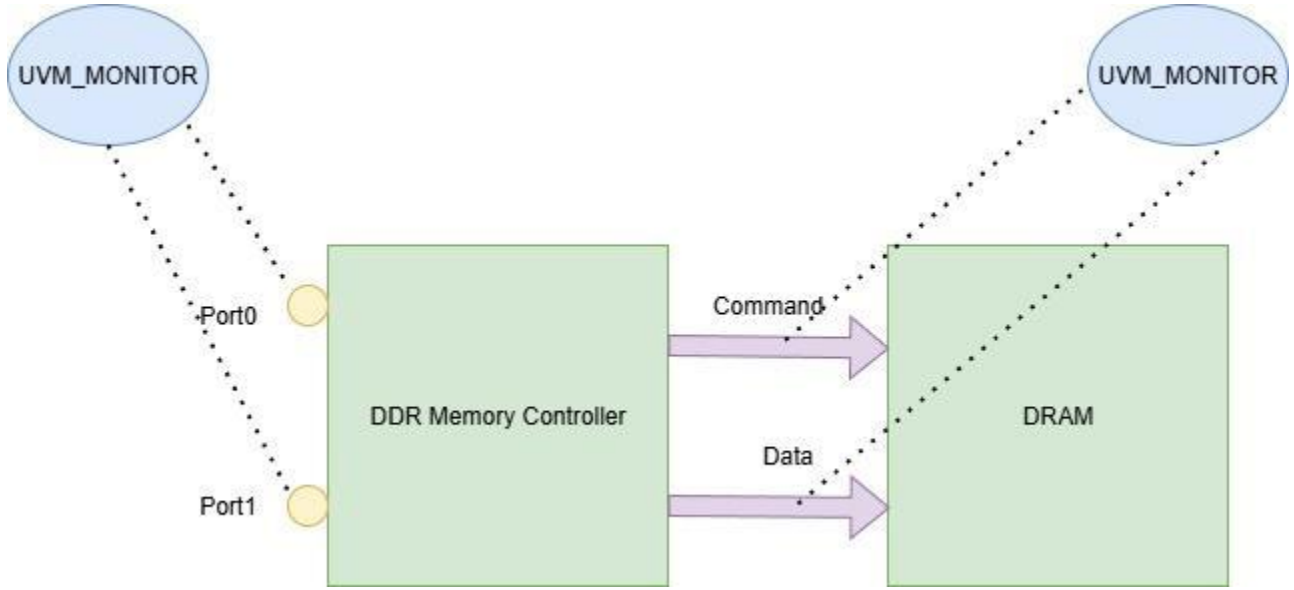
A Modern Debug Paradigm

The script can also parse the log file generated by the architectural model to generate similar visualizations that aid quicker debug convergence

Any verification team, that is doing functional, or performance verification can benefit from the ideas presented here to boost the debug efficiency

DDR Memory Controller's Testbench Overview

- This is a simplified representation of DDR Memory Controller's test bench
- We have UVM monitors at the input ports where read/write transactions come in and at the DRAM memory where they are written to & read from.
- The log files generated by these monitors serve as inputs to the Python-based visualization scripts, which process and render them into meaningful graphical representations
- For the rest of the presentation logs generated at the input ports are called input_port_*.csv and at the DRAM are called ddr5/lpddr5 tracker.csv
- The CSV format was intentionally selected for its simplicity and compatibility with Python-based data-processing workflows.



List of Python Scripts And Their Utilities

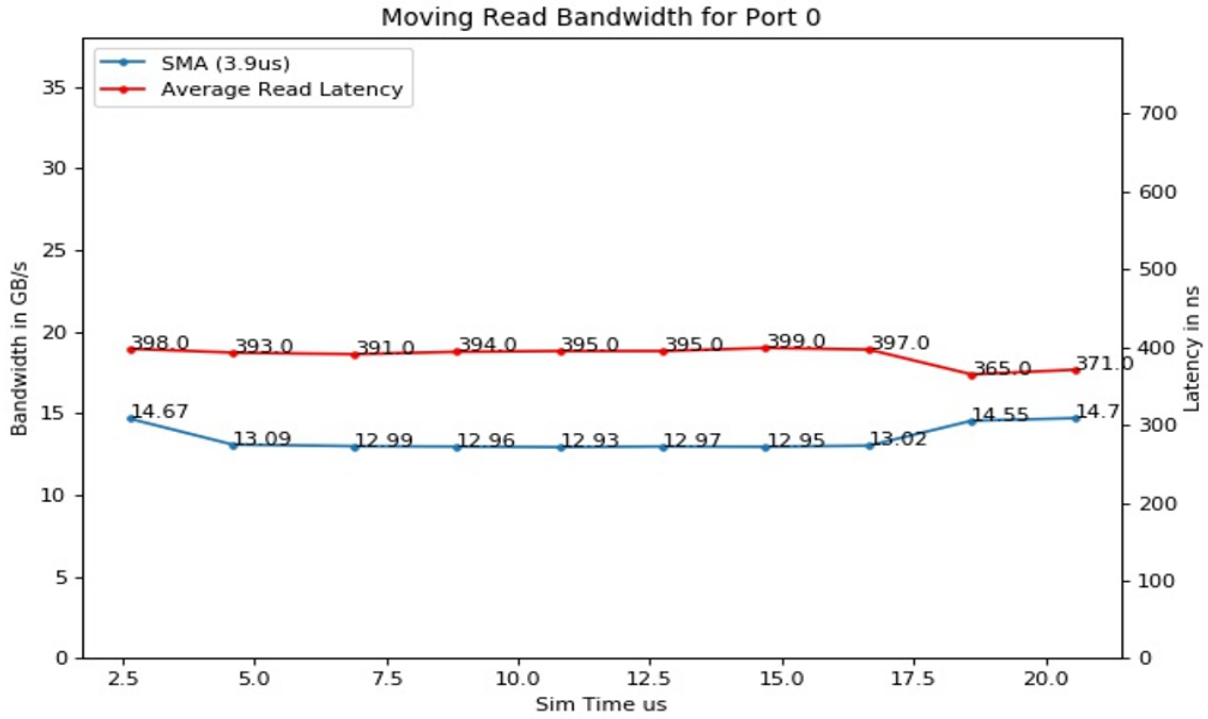
Script Name	Input File	Intent	Utility
plot_moving_bandwidth.py	input_port_*.csv files	Plots the simple moving average (SMA) of the read and write bandwidth seen on both the input ports over a pre-defined interval	Debugging failing test cases that report less than expected bandwidth; the time where a 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	Debugging read/write testcases where this plot can easily flag bandwidth imbalances among threads

List of Python Scripts And Their Utilities

Script Name	Input File	Intent	Utility
plot_per_bank_activity.py	ddr/lpddr tracker.csv files	Plots the distribution of critical DRAM commands per bank, bank group pair	Debugging a linear test case with bank group interleaving enabled wherein every thread is expected to open preset number of banks
dram_timings_analysis.py	ddr/lpddr tracker.csv 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

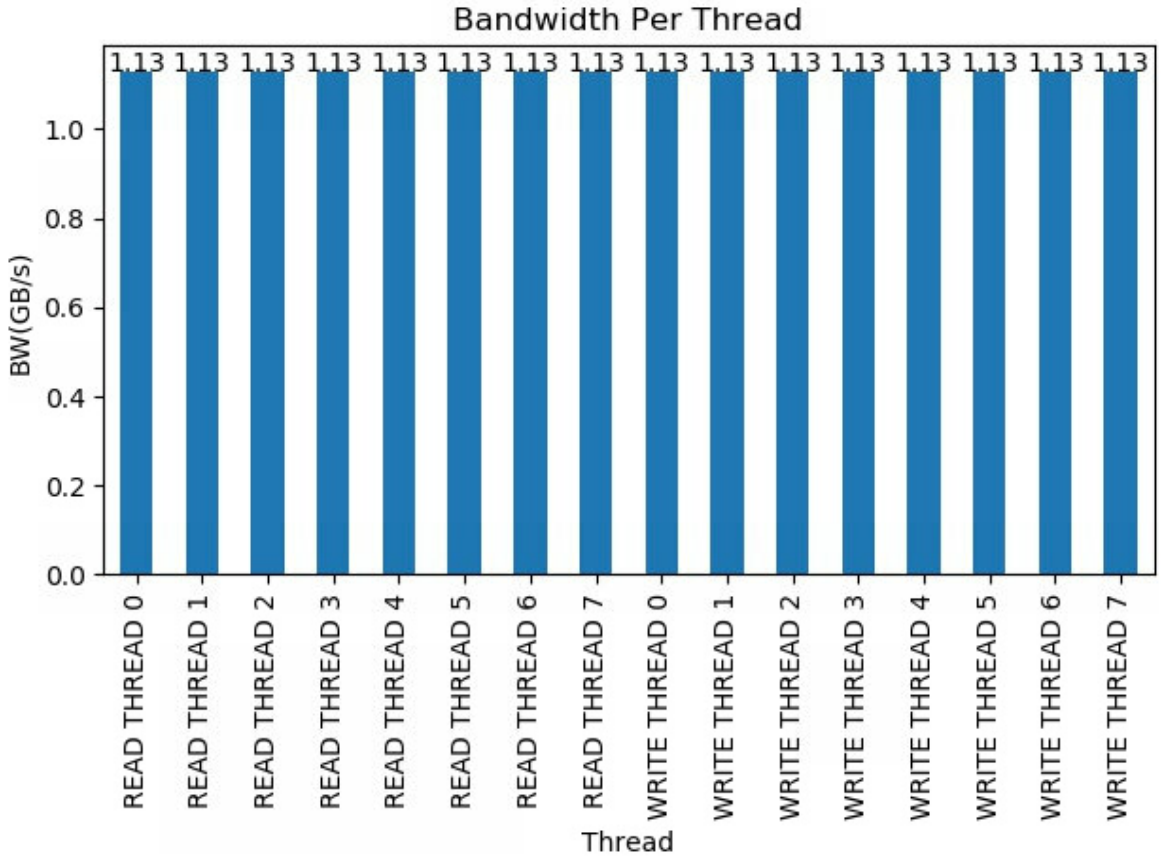
Sample Plot - SMA Of Latency and Bandwidth

- This figure shows a sample plot of small moving average (SMA) of read latency (shown in red) and bandwidth measured and plotted every 3.9us .
- 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 presentation are also reviewed before generating a waveform to get as much debug information as possible.



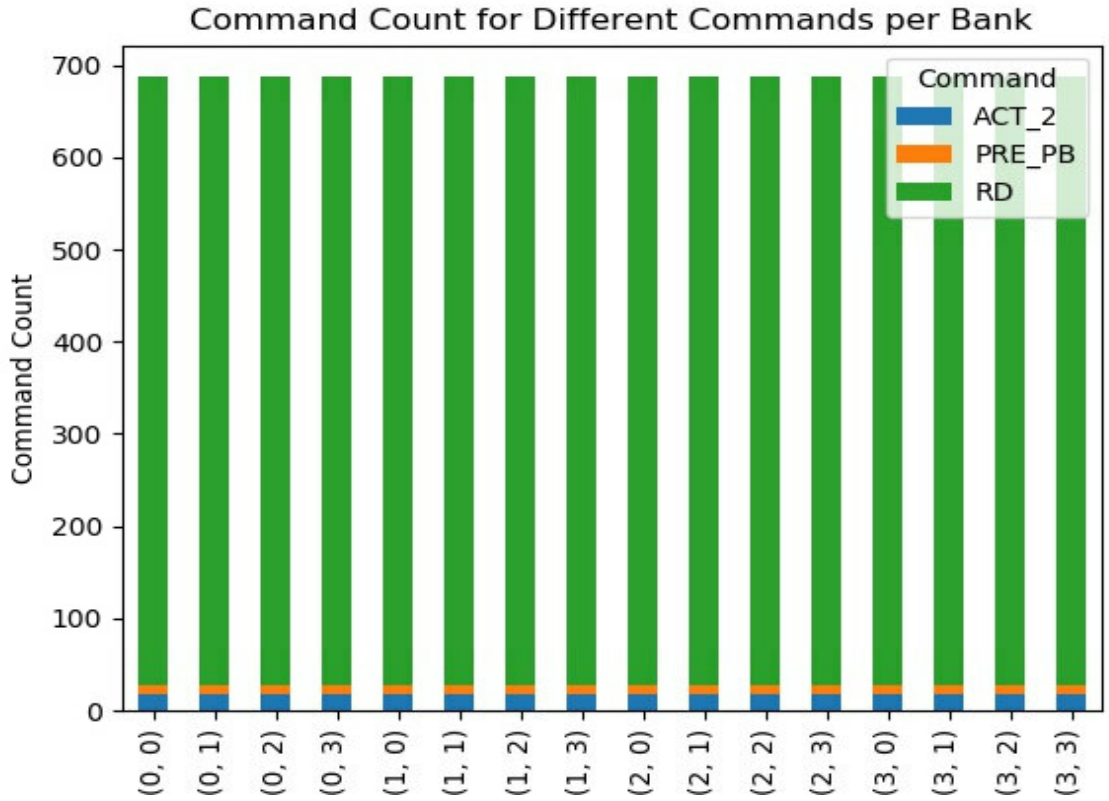
Sample Plot – Bandwidth Per Thread

- This plot is 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.
- Example plot is for an eight read/write 50 read /50 write linear test case run in LPDDR with two bank group interleaving.
- The test is expected to have some bank “fighting” since 16 threads each accessing two banks 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 trying to access it.
- Although bandwidth hogging is a possibility in this test, the plot clearly shows that the bandwidth is equally distributed among all masters.



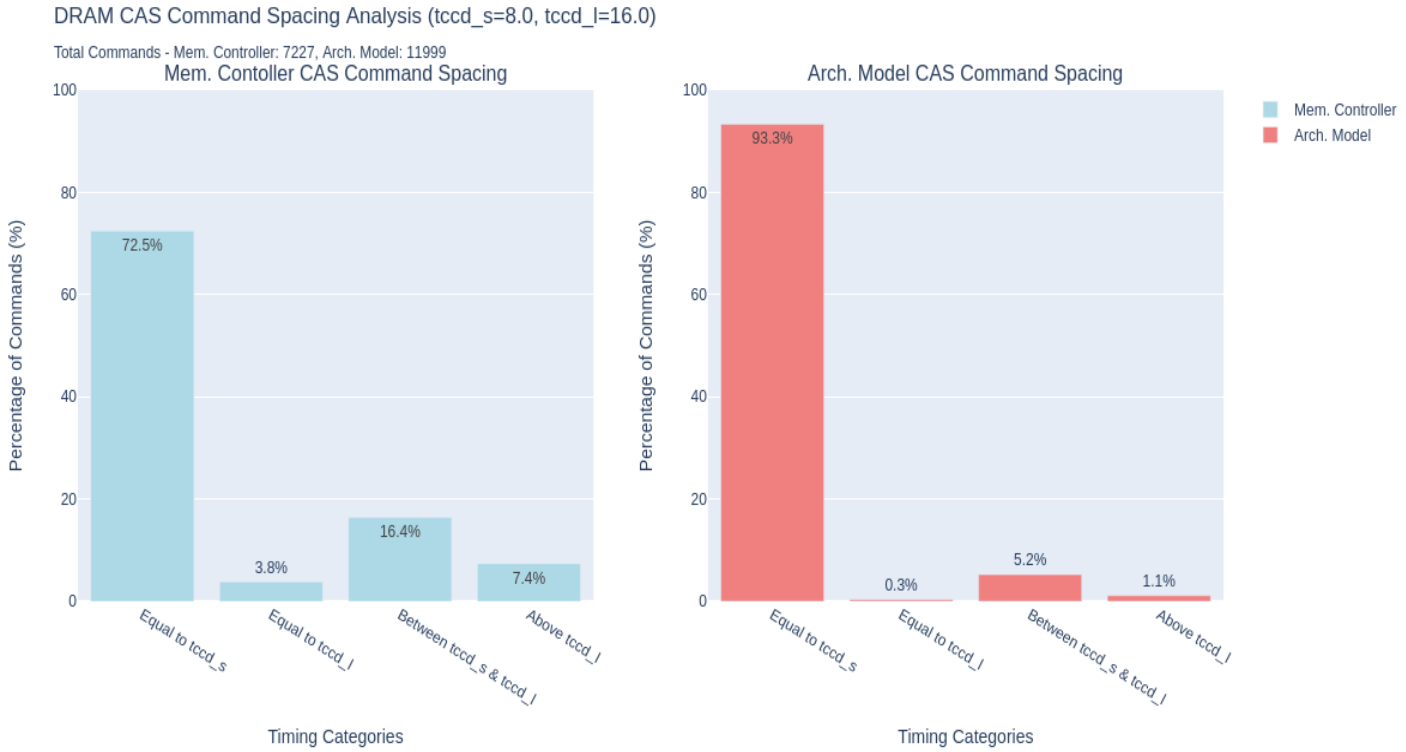
Sample Plot – Per Bank Activity

- The bank activity plot was developed primarily to identify if a test failure is due to incorrect address generation or an incorrect testbench setup issue.
- Example plot shows the distribution of Activate (ACT) , Precharge(PRE) and READ commands per bank group/bank pair for a read only test
- 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 incremental 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.
- Any deviations from this expected bank activity is easily spotted by viewing this plot.



Sample Plot – Safe Timing Histogram

- This plot 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 is a DDR incremental address 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

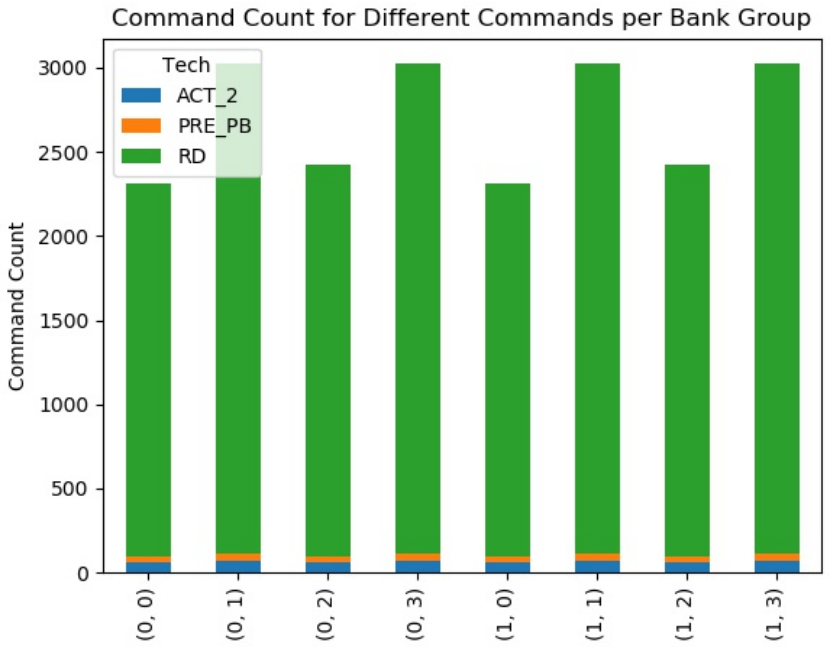


Example Debug Use Case For Plot Per Bank Activity

- Per bank activity plots are extremely useful to debug address access patterns that are designed to open a preset number of banks and bank groups
- In DDR the memory is organized as rows, columns, banks, bank groups and ranks
- Some test cases intentionally have bank group interleaving enabled to help the memory controller have parallelism while scheduling transactions to the DRAM memory
- Based on the bank group interleaving picked for a test we are already aware of the expected number of banks that has to have some read/wrote accesses.
- In these tests the plot_per_bank_activity comes very handy since it is easy to spot check if the expected number of banks opened matches the expectations or not
- For optimal performance we expect equal distribution of the CAS (command access strobe) commands in every bank

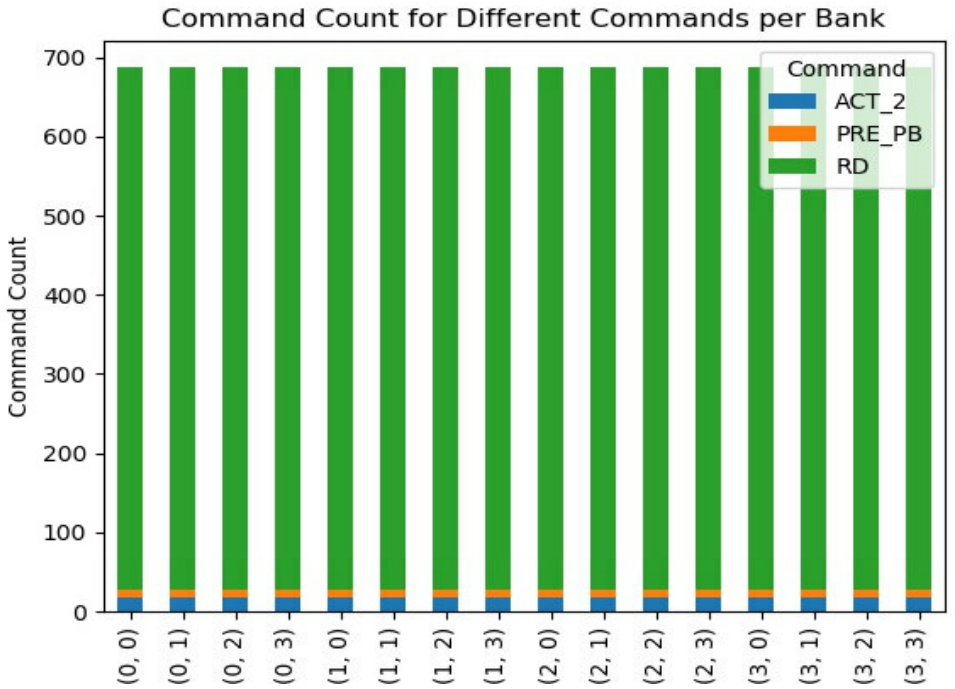
Example Debug Use Case For Plot Per Bank Activity

- Example plots on the right shows a test that reports unexpected bank activity
- This plot is a snippet from a subsystem incremental address 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
- Test reported poor performance numbers
- Issue was easily root caused to be a stimulus bug that generated incorrect addresses by viewing this plot
- Instead of waiting to generate waveforms that can take many hours these plots offer intuitive debug directions the engineer can take before getting a waveform



Example Debug Use Case For Plot Per Bank Activity

- Plot on the right shows the correct bank activity after the stimulus issue was fixed
- 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 per BG/BA pair



Conclusion

- 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 waveforms, thereby making them a 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.

COPYRIGHT AND DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


AMD