A Low-cost yet effective coverage model for fast functional coverage closure

Roshan Paul roshanp@nvidia.com, Koushik Ramakrishnan koushikr@nvidia.com,
Suresh M K smk@nvidia.com, Ajay Rupanagudi arupanagudi@nvidia.com,
Tathagato Bose tbose@nvidia.com, Guru Venkatesh K guruv@nvidia.com,
Vaidyanathan Sambasivan vsambasivan@nvidia.com, Subodh Prabhu subodhp@nvidia.com

Abstract-
Achieving rapid and efficient functional coverage closure remains a persistent challenge for large unit and cluster testbenches. Traditional coverage report generation and analysis methods are often slow, resource-intensive, and can delay verification closure. This paper introduces "Low Weight Coverage", a novel, cost-effective, and adequate methodology to accelerate functional coverage closure while maintaining quality. The proposed model addresses key challenges in existing coverage analysis, offering benefits on multiple fronts including improved stimulus quality, testcase ranking, and better submission quality, thereby enhancing the overall verification efficiency.

I. Introduction

In the early stages of the project lifecycle, coverage coding and analysis often receive low priority due to the time-intensive nature of report generation. For large unit and cluster testbenches, coverage reports may take up to a week to generate, leading to resource wastage and increased turnaround time for report generation, review, feedback collection, and corrective measures. Furthermore, enabling coverage in both design and testbench environments often necessitates the inclusion of extensive new coverage code, particularly from reusable design and verification IPs, which complicates and prolongs the compilation process. An additional challenge lies in the inefficient extraction and analysis of coverage contributions on a per-testcase basis.

II. Low Weight Coverage (LWC) Methodology

A. *LWC model overview*

The Low Weight Coverage (LWC) model aims to streamline the coverage process by integrating counters which we refer to as ‘stat counters’ implemented within the testbench and RTL, along with a post-regression parsing script as illustrated in Figure 1. Each ‘stat counter’ in the LWC model corresponds to a ‘coverage item’ and is incremented when a corresponding ‘coverage event’ occurs. This approach is analogous to invoking the ‘sample’ function within a functional cover-group. At the end of test all the stats (coverage scenarios) and the corresponding counter values are printed in a log file.

For example, consider number of ‘valid’ transactions. Some tests may not generate any ‘valid’ transactions, which can be acceptable. However, we expect a reasonable number of ‘valid’ transactions across an entire regression. This is the rationale behind using a counter instead of a functional cover-point.
This mechanism measures “real stimulus quality” rather than indicating what is covered and not. This allows us to assess whether a regression meets its ‘aggregate’ coverage goals without needing a coverage build/compile.
After a regression completes, the ‘lwc_post_regression_parser’ reads the stat counter values of each test, calculates the average for each stat across all tests, and compares them against expected thresholds provided in a separate file.

The “low weight” jargon comes from the idea that this is a cheap way of getting coverage due to its zero impact on compile time, very low runtime impact, instant coverage report generation, and because it doesn’t provide all the features that functional coverage provides. In contrast, functional coverage enabled compilation, test runtime, and report generation can be much slower. This is common in large simulation enviroments containing multiple instances of RTL modules, UVM environments, shared design and VIPs where substantial coverage code is included.

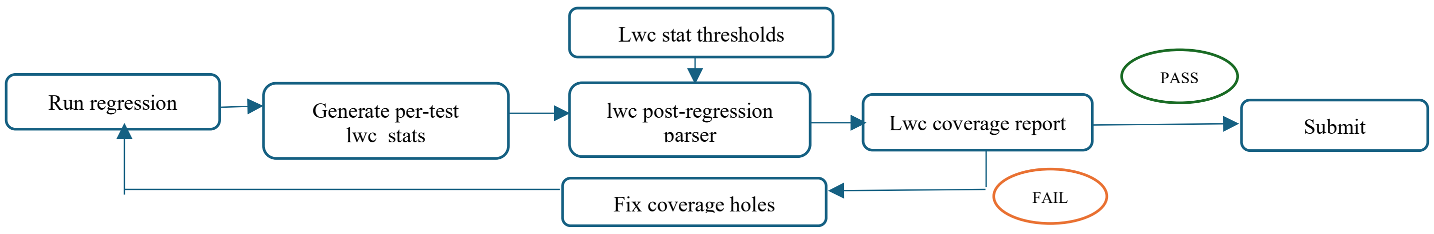
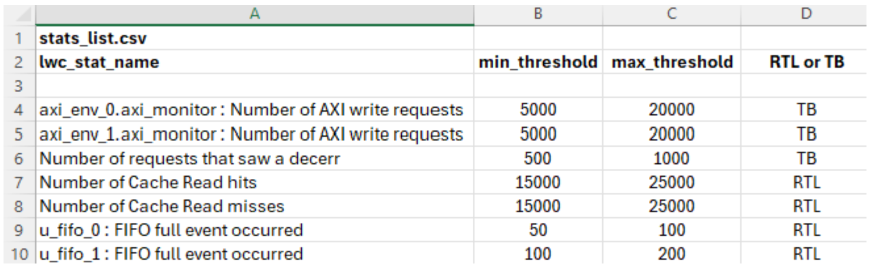


Figure 1. Low weight coverage flow

B. *How are LWC counters Implemented?*

When we decide to add a new stat/ cover item, we first add it to the ‘stats_list.csv’. Figure 2 describes this file containing columns ‘lwc_counter_name’, ‘min_threshold’, and ‘max_threshold’.



	A	B	C	D
1	**stats_list.csv**			
2	**lwc_stat_name**	**min_threshold**	**max_threshold**	**RTL or TB**
3				
4	axi_env_0.axi_monitor : Number of AXI write requests	5000	20000	TB
5	axi_env_1.axi_monitor : Number of AXI write requests	5000	20000	TB
6	Number of requests that saw a decerr	500	1000	TB
7	Number of Cache Read hits	15000	25000	RTL
8	Number of Cache Read misses	15000	25000	RTL
9	u_fifo_0 : FIFO full event occurred	50	100	RTL
10	u_fifo_1 : FIFO full event occurred	100	200	RTL

Figure 2. stats_list.csv

First three scenarios from Figure 2 are implemented in the testbench and the remaining four in the RTL.

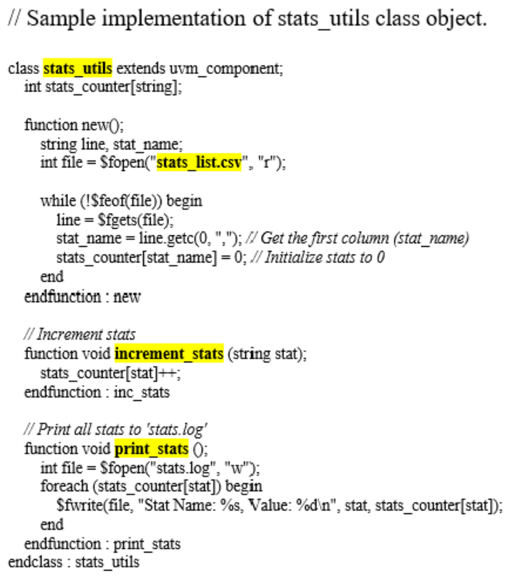
I) Testbench stat counter implementation:

Testbench contains a class object “stats_utils” which will read the ‘stats_list.csv’ file, load the stat counter names marked TB into an associative array, and then initialize them to 0. During runtime, the counters can be incremented from any part of the testbench. TB components increment a stat counter by calling the “increment_stats” function when the corresponding scenario occurs. This is like calling a functional cover-group’s ‘sample’ function when the scenario to be covered is detected. Stat counters can also be incremented from re-usable VIPs either with direct access to stats_utils or via callbacks depending on the level of support available. At end of the test, all stat names and values are printed in “stats.log” (Figure 3).

```
COVER_INFO_TB : Number of requests that saw a decerr =  600
COVER_INFO_TB : uvm_test_top.top_env.axi_env_0.axi_monitor : Number of AXI write requests = 100
COVER_INFO_TB : uvm_test_top.top_env.axi_env_1.axi_monitor : Number of AXI write requests = 150
```

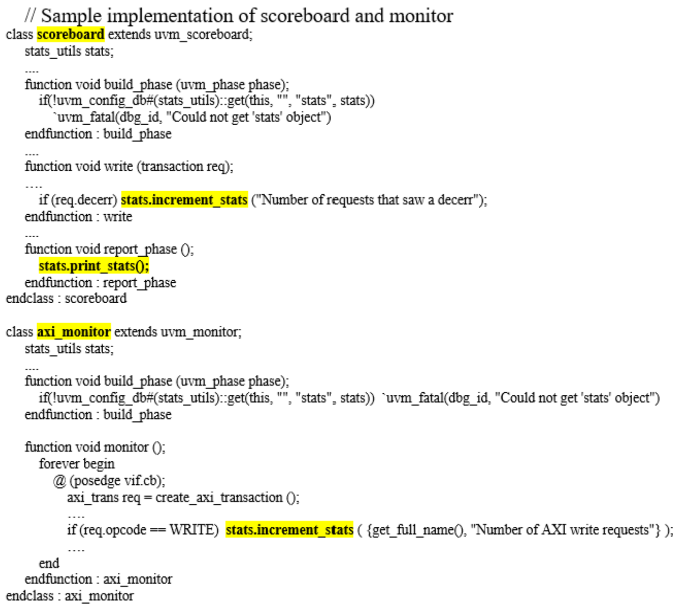
Figure 3. stats.log output in the test results directory

Code snippet in Figure 4 show the definition of “stats_utils” class and Figure 5 shows the way ‘increment_stats’ function is called. NOTE: All the code snippets provided in this paper have been edited to convey the intent concisely.



```
// Sample implementation of stats_utils class object.

class stats_utils extends uvm_component;
   int stats_counter[string];

   function new();
      string line, stat_name;
      int file = $fopen("stats_list.csv", "r");

      while (!$feof(file)) begin
         line = $fgets(file);
         stat_name = line.getc(0, ","); // Get the first column (stat_name)
         stats_counter[stat_name] = 0; // Initialize stats to 0
      end
   endfunction : new

   // Increment stats
   function void increment_stats (string stat);
      stats_counter[stat]++;
   endfunction : inc_stats

   // Print all stats to 'stats.log'
   function void print_stats ();
      int file = $fopen("stats.log", "w");
      foreach (stats_counter[stat]) begin
         $fwrite(file, "Stat Name: %s, Value: %d\n", stat, stats_counter[stat]);
      end
   endfunction : print_stats
endclass : stats_utils
```

Figure 4. stats_utils class definition



```
   // Sample implementation of scoreboard and monitor
class scoreboard extends uvm_scoreboard;
   stats_utils stats;
   ....
   function void build_phase (uvm_phase phase);
      if(!uvm_config_db#(stats_utils)::get(this, "", "stats", stats))
         `uvm_fatal(dbg_id, "Could not get 'stats' object")
   endfunction : build_phase
   ....
   function void write (transaction req);
   ....
      if (req.decerr) stats.increment_stats ("Number of requests that saw a decerr");
   endfunction : write
   ....
   function void report_phase ();
      stats.print_stats();
   endfunction : report_phase
endclass : scoreboard

class axi_monitor extends uvm_monitor;
   stats_utils stats;
   ....
   function void build_phase (uvm_phase phase);
      if(!uvm_config_db#(stats_utils)::get(this, "", "stats", stats))  `uvm_fatal(dbg_id, "Could not get 'stats' object")
   endfunction : build_phase

   function void monitor ();
      forever begin
         @ (posedge vif.cb);
            axi_trans req = create_axi_transaction ();
            ....
            if (req.opcode == WRITE)  stats.increment_stats ( {get_full_name(), "Number of AXI write requests"} );
            ....
      end
   endfunction : axi_monitor
endclass : axi_monitor
```

Figure 5. Scoreboard implementation

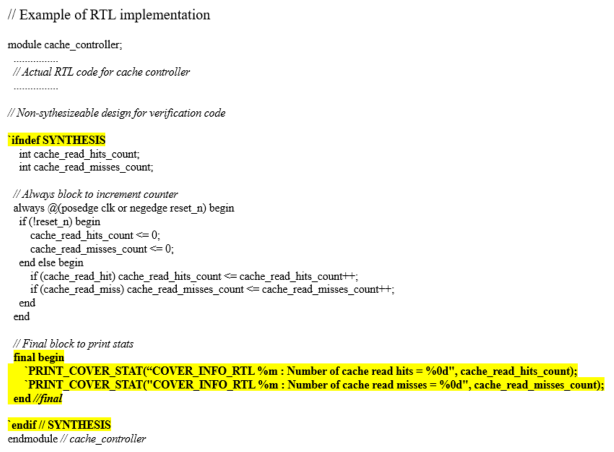
II) RTL stat counter implementation:

The coverage scenarios are implemented in the RTL as non-synthesizable counters. These counters can be incremented from a module, interface, or reusable design IPs. The RTL stat counter values are printed in the same 'stats.log' file as described in Figure 6.

```
COVER_INFO_RTL : u_top.u_cache_top.u_cache_ctrl : Number of Cache Read hits             = 115
COVER_INFO_RTL : u_top.u_cache_top.u_cache_ctrl : Number of Cache Read misses            =  82
COVER_INFO_RTL : u_top.u_cache_top.u_cache_ctrl.u_fifo_0 : FIFO full event occurred       =   1
```

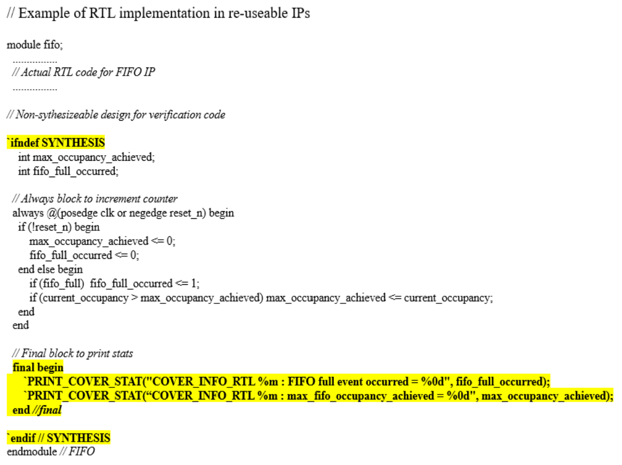
Figure 6. stats.log output in the test results directory

Figure 7 and Figure 8 describe RTL counters implemented in a module and a shared IP respectively.



```
// Example of RTL implementation

module cache_controller;
  ................
  // Actual RTL code for cache controller
  ................

// Non-sythesizeable design for verification code

`ifndef SYNTHESIS
   int cache_read_hits_count;
   int cache_read_misses_count;

  // Always block to increment counter
  always @(posedge clk or negedge reset_n) begin
   if (!reset_n) begin
      cache_read_hits_count <= 0;
      cache_read_misses_count <= 0;
   end else begin
      if (cache_read_hit) cache_read_hits_count <= cache_read_hits_count++;
      if (cache_read_miss) cache_read_misses_count <= cache_read_misses_count++;
   end
  end

  // Final block to print stats
  final begin
    `PRINT_COVER_STAT("COVER_INFO_RTL %m : Number of cache read hits = %0d", cache_read_hits_count);
    `PRINT_COVER_STAT("COVER_INFO_RTL %m : Number of cache read misses = %0d", cache_read_misses_count);
  end //final

`endif // SYNTHESIS
endmodule // cache_controller
```

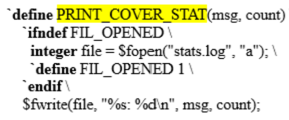
Figure 7. Cache controller counter implementation



```
// Example of RTL implementation in re-useable IPs

module fifo;
  ................
  // Actual RTL code for FIFO IP
  ................

// Non-sythesizeable design for verification code

`ifndef SYNTHESIS
   int max_occupancy_achieved;
   int fifo_full_occurred;

  // Always block to increment counter
  always @(posedge clk or negedge reset_n) begin
   if (!reset_n) begin
      max_occupancy_achieved <= 0;
      fifo_full_occurred <= 0;
   end else begin
      if (fifo_full)  fifo_full_occurred <= 1;
      if (current_occupancy > max_occupancy_achieved) max_occupancy_achieved <= current_occupancy;
   end
  end

  // Final block to print stats
  final begin
    `PRINT_COVER_STAT("COVER_INFO_RTL %m : FIFO full event occurred = %0d", fifo_full_occurred);
    `PRINT_COVER_STAT("COVER_INFO_RTL %m : max_fifo_occupancy_achieved = %0d", max_occupancy_achieved);
  end //final

`endif // SYNTHESIS
endmodule // FIFO
```

Figure 8. FIFO design IP counter implementation

RTL uses the macro "PRINT_COVER_STAT" shown in Figure 9 to print the stats to a file.



```
`define PRINT_COVER_STAT(msg, count) \
   `ifndef FIL_OPENED \
    integer file = $fopen("stats.log", "a"); \
    `define FIL_OPENED 1 \
   `endif \
   $fwrite(file, "%s: %d\n", msg, count);
```

Figure 9. Macro that writes counter values to the file

When a test finishes, its simulation results directory will contain a 'stats.log' file. This shows the amount of coverage achieved by that test. Once the entire regression is done the 'lwc_post_regression_parser' reads each test's "stats.log" and computes the 'average' for each stat across the regression. This is compared against the min and max thresholds in "stats_list.csv".

In the 'stats.log' example output, "FIFO full event occurred" being 1 only indicates that this instance of the test filled up the FIFO but that doesn't necessarily mean the overall coverage expectations have been met.

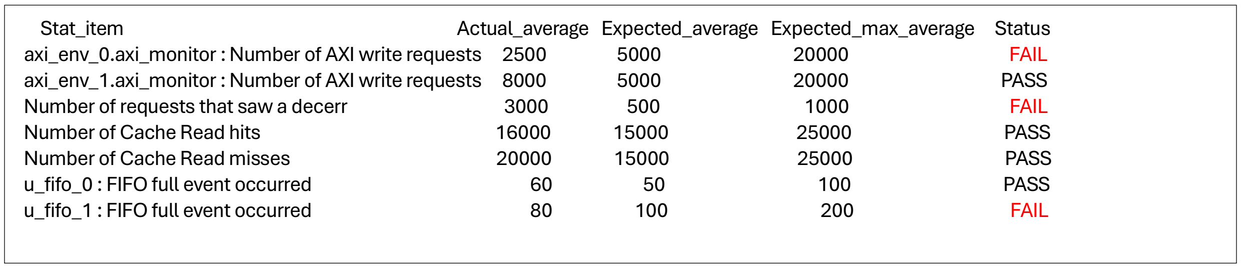
Let's say a regression running 1000 tests saw an average count of 60 (number of times full event occurred) for "u_fifo_0 : FIFO full event occurred" and 30 for "u_fifo_1 : FIFO full event occurred".

"u_fifo_0" falls in the threshold range since "min_threshold=50" and "max_threshold=100" and hence has met the coverage goals. However, "u_fifo_1" count of 30 falls outside the threshold range of "min_threshold=100" and "max_threshold=200". This could indicate that the stimulus is lacking, and the design isn't properly tested.

The user must set the thresholds to 'reasonable' and appropriate values for each stat/cover scenario.

C. *LWC Post-Regression Parser*

The “lwc_post_regression_parser” script is activated once the regression process is complete and the individual “stats.log” files, which contain coverage values for each test, are available. The parser reads these log files, calculates the average coverage achieved, and compares it to the threshold values specified in “stats_list.csv”. Based on this comparison, it generates a PASS or FAIL report (Figure 10). A FAIL status is given when actual coverage falls outside the expected range, indicating that the desired level of stimulus coverage has not been met.



```
   Stat_item                                      Actual_average  Expected_average  Expected_max_average   Status
axi_env_0.axi_monitor : Number of AXI write requests   2500          5000              20000                FAIL
axi_env_1.axi_monitor : Number of AXI write requests   8000          5000              20000               PASS
Number of requests that saw a decerr                    3000           500               1000                FAIL
Number of Cache Read hits                              16000         15000              25000                PASS
Number of Cache Read misses                            20000         15000              25000                PASS
u_fifo_0 : FIFO full event occurred                       60            50                100                PASS
u_fifo_1 : FIFO full event occurred                       80           100                200                FAIL
```

Figure 10. Sample ‘lwc_regression_cov_report.log’ output of post-regression parser script

D. *Testcase ranking*

Another feature of the LWC model is, that it computes specific metrics to assess and rank the efficacy of each test case, facilitating detailed coverage contribution analysis on a per-testcase basis. This is achieved by assigning metrics like ‘score’ and ‘rarity’ to each test.

‘Score’, ‘rarity’, ‘volume’, ‘breadth’ are similar concepts as introduced in [1].

Metrics used by LWC –

Score – how valuable a test is for bulk verification.

$$Score = (Volume^2 + Rarity\_Factor * Breadth^2 - Time\ Factor - Time^2)^{Power\_Factor}$$

Rarity – how crucial a test is for rare scenario verification. Measured as 30% rarest buckets based on [1]

SPF (score per farm cycle) – how efficient a test is for bulk verification (for a given farm time).

$$SPF=(Score*TimeFactor)/cpuTime$$

RPF (rare score per farm cycle) – how efficient a test is for rare scenario verification (for a given farm time).

$$RPF=(RareScore*TimeFactor)/cpuTime$$

Figure 11 shows an example graph of test ranking.
Tests in green scored the most and tests in red scored the least.

Breadth (X-axis) – tests hitting variety of coverage i.e unique stat hits
Volume (Y-axis) – test hitting bulk of your coverage i.e number of all stat hits.

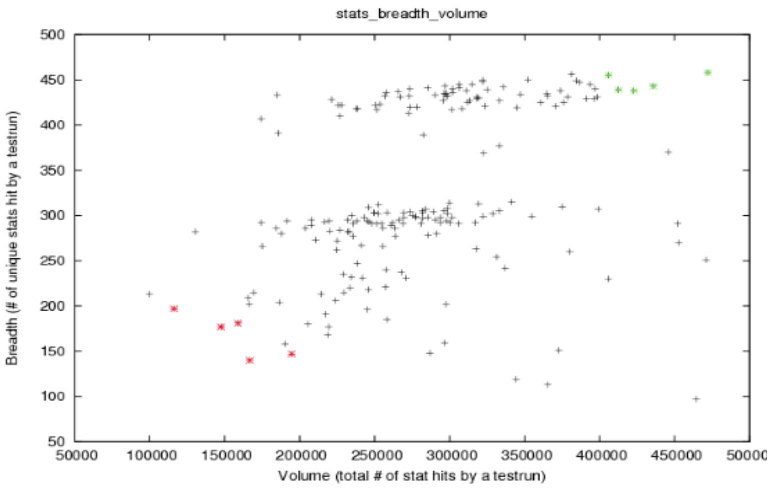


Figure 11. Test ranking – Breadth vs Volume

The “lwc_post_regression_parser” script does test ranking. While parsing, the script keeps track of unique and bulk ‘stats’ and CPU-time in-order to calculated metrics like Volume, Breadth, rarity, SPF, RPF and the total score. These metrics are then used to create graphs for studies, statistics, and comparisons.

III. Impact of LWC Model On The Verification process

A. *Bugs caught by LWC*

The primary motivation behind the LWC approach was a bug discovered at our cluster testbench where all the requests issued to the cluster were getting dropped due to 'decerrs' (decode errors). The bug wasn't caught by the individual IP scoreboards, or the 'basic' self-checking directed tests and randoms used for the initial bring-up.

The LWC approach was proposed to identify such stimulus issues much earlier in the project timeline.
Since its deployment, LWC has caught many RTL, testbench, stimulus, and even architecture bugs.

Examples of bugs caught by LWC

a) An RTL issue was found during the bring-up of the set associative cache where only 32 of the total 64 sets were being accessed. Rest of the sets were unreachable. DV scoreboard doesn't model set ID calculations as they only affect performance and not the functionality. This bug would typically be found either during performance verification or coverage reviews. Both happen later in the project timeline. LWC was able to catch this issue immediately as stats were added to report the number of accesses to each set in the cache.
b) A bug in the ECC 'algorithm' was discovered when too many 'correctable' errors were detected as 'uncorrectable' errors. RTL and Scoreboard model the same 'algorithm'; hence, verification couldn't catch the bug. The 'algorithm bug' was discovered when LWC's "max_threshold" check failed.
c) Bug found in the RTL cache read hit logic while investigating low stat hit counts once the stat was implemented.
d) RTL bugs caught after LWC post-regression checks complained of missing cross-feature interaction. In this case initial feature bring-up with the regressions passed. But once stat counters were added for cross-feature-interaction, the post-regression parser identified corner case scenarios, missing stimulus, and hence bugs hiding due to them.
e) A bad test-plan change resulted in silent disablement of a feature. This was caught during regressions immediately.
f) A new opcode that was added to CHI protocol BFM (which is shared across many IPs). But a BFM bug resulted in the opcode being not sent in certain cases. This was caught after adding a stat counter in the BFM's 'consumer testbench'. The stat counter was later moved to the BFM to make it available to all consumer teams.
g) A bad testcase change resulted in a high amount of illegal traffic. This caused a subtle drop in coverage which was not enough to push it beyond the threshold range as only a single test was affected. The issue was eventually caught by our anomaly detection code which investigates any drops in coverage during week-to-week coverage score comparisons. Anomaly detection is explained further in a later section.

C. *In-Simulation and Post-Simulation Checks*

LWC has been instrumental in making sure directed tests are meeting their intent. Often we develop directed tests to achieve specific scenarios for coverage purposes that may not necessarily occur in random tests. They may be targeting a corner case that is hard to hit or attempting to hit the depths of storage structures. We have used these stats counters to perform self-checks to ensure the scenarios are hit. This is done either by accessing the testbench stat counter in-simulation or performing a post-simulation check using our stats log parser.

For example, if a directed test is written to target u_fifo_1, the log parser will ensure "u_fifo_1 : FIFO full event occurred = 1" is seen in the stats log for this test. With this self-check, the testcase doesn't get outdated as the project progresses, and RTL changes.

D. *Reuse at Higher Level Testbenches*

Most cluster and SOCV testcases are written as directed tests with the intention to hit specific use cases.

However, these testbenches may not necessarily have the modelling or visibility into the individual IP designs or testbenches. The simple nature of counters implemented within a testbench, or RTL make it an easy candidate for re-use. Our LWC counters are frequently used by higher level testbenches to make sure the intended features are being tested. This has saved a lot of debugging and bringing up time for cluster/SOCV testbenches.

E. *Self-Correcting Stimulus*

The object-oriented nature of the stats counters make it easy for a testcase to get live in-simulation coverage feedback. This can used to cover the depths of complex memory structures which can be hard to achieve. Typically, these are done by adding complex constraints with partial modeling of memory structures in constraints. Risks include over-constraining or incorrect modelling in the testcase. Access to prior coverage information can be useful but SV cover-groups don't provide coverage feed-back with the ease that LWC approach can. Future work focuses on leveraging LWC counters for the application of reactive stimulus.

F. *Testcase Tuning*

The test ranking feature of LWC is used to identify the test cases that are more valuable and efficient at hitting coverage than others. The graphs below show how testcase tuning was done for an IP throughout a project.
Average "lwcBreadth" (Figure 12) has mainly remained the same whereas "lwcVolume" (Figure 13) has decreased. However, "lwcSpf" and "lwcRpf" (Figure 14) have increased. This indicates that tests have become shorter and more efficient at consuming farm resources.
The drop in "lwcVolume" i.e bulk coverage is acceptable. "lwcBreadth" staying the same means the tests are still hitting the same diverse coverage.

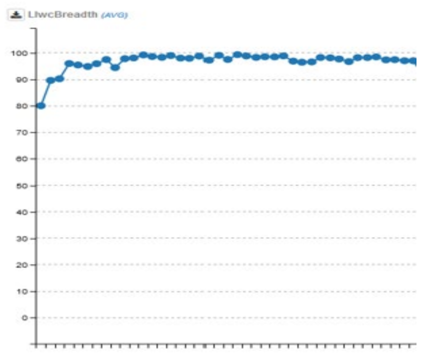


Figure 12. LwcBreadth over project lifecyle

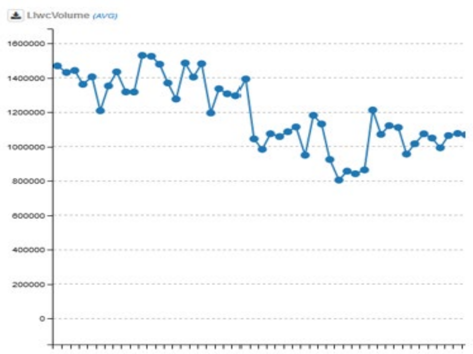


Figure 13. LwcVolume over project lifecyle

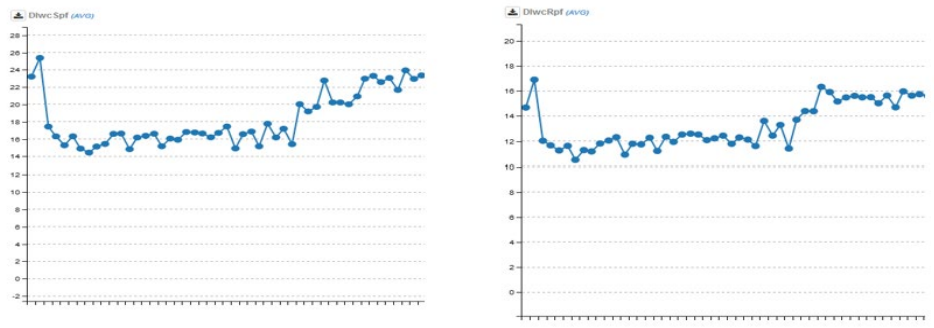


Figure 14. LwcSpf (score per farm cycle) and LwcRpf (rare score per farm cycle)

G. *Impact to the verification teams*

LWC model provides an automated and near-instant coverage feedback which may not be possible with traditional functional coverage. This saves us any manual efforts involved in running a coverage enabled build and regression, generating a merged functional coverage report. This has increased the verification productivity by finding significant bugs early in the development cycle.

What has the LWC methodology allowed us to do ?

a) Enabled LWC methodology in all IP, sub-IP, cluster, and SOCV testbenches across our entire division.
b) Used LWC results to qualify every change before submission.
c) Implemented stat counters in re-usable design IPs and verification IPs.
d) Used LWC as the go-to source of coverage closure for feature verification.
e) Provided us a cheap and consistent way to measure how efficient regressions are.
f) Used stat counters to report metrics, for example, evolution of static latencies per IP over the course of a project.
g) Allowed project-to-project / milestone-to-milestone / IP-to-IP "score" comparisons.
h) Use "anomaly detection" to keep track of any drops in week-to-week coverage scores and notify the user.

The graphs in Figure 15 show project-1 as being the mature project with a stable score (in terms of lwcBreadth). Towards the end of the project lwcVolume reduces as the tests become more efficient.

Project-2 (Figure 16) is the newer project where, coverage score starts close to 0 and shows many fluctuations on the way up. lwcVolume increases as each test starts hitting more coverage as project progresses.

The anomaly detector can catch huge drops (circled in project-1) and subtle drops (circled in project-2) in the scores without relying on user specified thresholds.

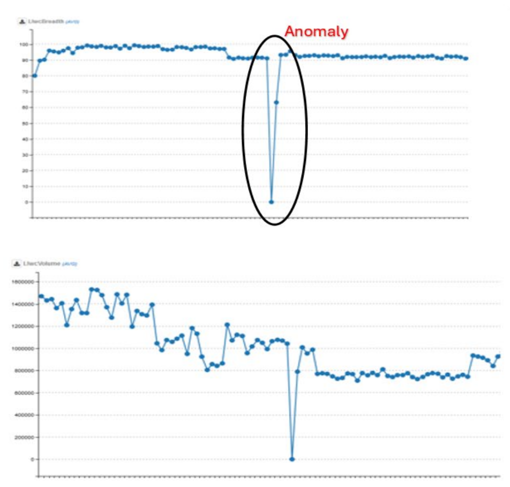


Figure 15. Project-1
lwcBreadth (top)
lwcVolume (bottom)

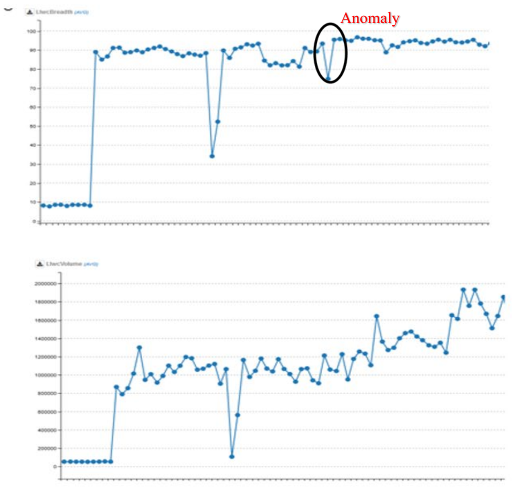


Figure 16. Project-2
lwcBreadth (top)
lwcVolume (bottom)

IV. Limitations of LWC

While the LWC model offers significant advantages, it is not a complete replacement for traditional functional coverage. LWC's use of simple counters as coverage bins limits its ability to handle complex cross-coverage scenarios, which can involve numerous cross-coverage bins along with features like ignore and illegal bins. Therefore, LWC must be used in conjunction with traditional functional coverage mechanism.

V. Performance metrics

We evaluated the impact of the LWC counter implementation by conducting comparative studies of regression runs with and without counters. The testing was performed using 10000 stat counters across various testbenches. Analysis was done using the VCS simulator.

For small testbenches, the introduction of counters resulted in an average runtime increase of 10%. This mainly due to logging the counter information. For large testbenches, the impact on runtime was 4%. There were no noticeable build/compile time deltas.

In Figure 17, the blue line being higher than the green line shows the runtime overhead of an LWC enabled run. Orange line and green being identical shows that LWC has no build/compile time impact.

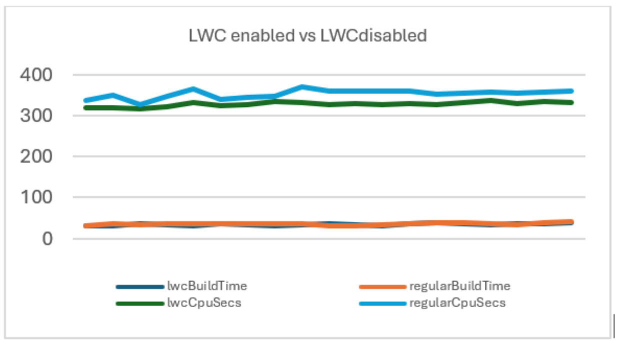


Figure 17. Build and runtime comparison of an LWC enabled run to an LWC disabled run

Figure 18 illustrates the comparison results of an LWC enabled build to a functional coverage enabled build. Comparison was done using 10000 cover-points in an LWC disabled build/run and 10000 stat counters in a Functional coverage disabled build.
LWC provides a 40-50% reduction in build time compared to a coverage enabled build and an average 20% in runtime savings.

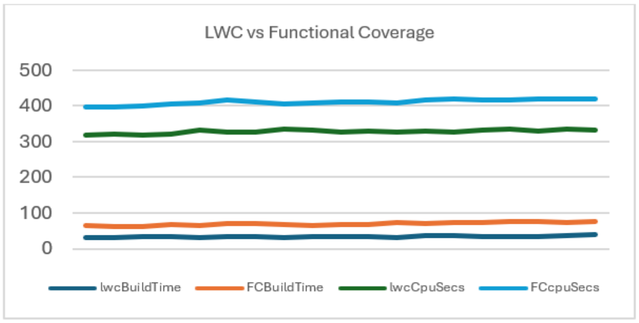


Figure 18. Build and runtime comparison of an LWC enabled run to Functional coverage enabled run

Figure 19 shows the comparison results of low weight coverage "scores" for the same codebase between two simulators. They have identical scores across long time spans. This helps validate LWC as a reliable methodology.

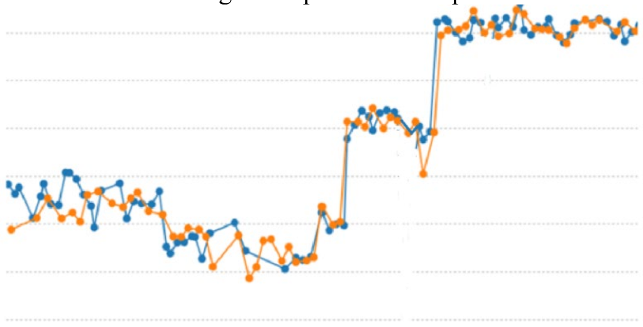


Figure 19. Coverage score comparison of "simulator-1" vs "simulator-2" over project cycle

VI. Future Work

Future work focusses on integrating LWC with LLM. This will use testcase ranking metrics to auto-tune a regression by injecting or removing test-cases until a specific coverage goal (lwcBreadth and lwcVolume) and efficiency (lwcSPF and lwcRPF) is met (this is a manual process today).

VII. Conclusion

The Low Weight Coverage (LWC) model presents a compelling approach to enhancing functional coverage closure in large unit and cluster testbenches. Its ability to provide near-instant coverage feedback, test ranking and prioritization, and enforce stimulus quality makes it a valuable tool for accelerating coverage closure.

References

[1] "Optimizing Random Test Constraints Using Machine Learning Algorithms", ARM. https://dvcon-proceedings.org/wp-content/uploads/optimizing-random-test-constraints-using-machine-learning-algorithms-presentation.pdf
[2] "Testbench coverage", Synopsys, https://www.synopsys.com/verification/simulation/vcs.html