# So you think you have good stimulus:
## System-level distributed metrics analysis and results

Andreas Meyer
Mentor Graphics Corp., Wilsonville, OR USA
andy_meyer@mentor.com

Alan Hunter
ARM, Austin TX, USA
Alan.Hunter@arm.com

*Abstract* - **Developing and maintaining an effective and efficient verification suite for a complex system requires the ability to measure, understand, and improve the environment. Distributed, hierarchical caches are an example of interacting components within an SoC. Understanding how well the components are verified is a challenge since the cache interactions are complex, the components are distributed across an environment, and the data is spread across one or more regressions. This paper discusses the challenges of collecting metrics, providing the visualization to understand complex state machine interactions, and then reviews results of a regression analysis.**

**A complex ARM-based SoC with multiple processors, and a coherent distributed multi-level cache is the basis of our study. A modern constrained-random test suite is used to generate regression suites, with traditional code and functional coverage methods used to steer and grade the test suite. While this approach has been successful, it does require significant compute resources to reach coverage closure, and it has been difficult to determine how to improve the efficiency and quality of the test suite.**

**We introduce statistical coverage as an approach to provide new coverage analysis capability. Within our ARM SoC project, we show how we are able to find and fix significant weaknesses in the stimulus that could not be seen using traditional code and functional coverage metrics. The stimulus improvements provided improved regression efficiency, found areas that had not been fully explored, and as a result found additional RTL issues.**

## 1. Introduction

Developing effective stimulus in an SoC environment, with complex interactions between large IP blocks can be challenging. Current methods for verification are not keeping up with complexity trends in modern SoCs such as ARM processor environments.

A common verification approach of constrained-random stimulus generation relies on coverage metrics to determine completeness, and provide the feedback needed to modify the constraints for full coverage. Current coverage methods are well suited for pointing out areas of code that haven't executed, or how well all branches of a state machine have been exercised. While existing coverage tools can even work reasonably well at measuring across a few pre-defined areas, there are currently no standard methods to capture interaction coverage of state machines distributed across multiple IP blocks.

Metrics have been used for some time to gather more detailed information on many aspects of verification performance and effectiveness. We have applied the approach outlined earlier [1] to generate a new view into the performance of an SoC stimulus suite.

Using real-world SoC development projects implementing state-of-the-art constrained random stimulus, we compared conventional coverage methods with new statistical coverage methods. We looked at example reports from the statistical coverage tool, the types of system-level information that it can report, and what the statistical coverage told us about the existing constrained-random test suite.

The new measurement capability gave us information on where the stimulus constraints needed improvement, in addition to uncovering some surprisingly trivial bugs. With new coverage information, it was possible to modify the stimulus constraints and rerun the test suite. We'll show the results from running the improved stimulus, including examples of new functional errors that were uncovered, and statistical coverage reports showing the coverage improvements and a corresponding higher density of system-level interactions within the system.

## 2. Project Overview

Our metrics experiments are based on a current dual-cluster ARM SoC project. The project uses a modern SystemVerilog based verification flow. Stimulus is generated through a carefully designed and maintained set of constraints, which use code and functional coverage metrics to determine completion criteria. This approach has been used successfully in a variety of projects, and at many integration levels from unit to system.

We instantiated statistical coverage in this existing environment, that had already closed code and functional coverage project goals, as a way to gain new insight into the verification environment by measuring system-level statistical coverage. As an example, functional coverage is excellent at checking that state machines have hit interesting points, or that FIFOs have filled and emptied. These are important for checking coverage of unit-level tests. At the system level, we are interested in checking that interactions between large IP blocks are taking place. This includes looking at cache coherence between IP blocks, or tracking memory or packet transfers between blocks. Our goal with statistical coverage is to gain new insight into the inter-IP operations at subsystem or system level.

Even though our goal was to look at system-level interactions, when we plugged statistical metrics into our environment, we were quite surprised at the range of unit and system-level information we received.

## 3. Approach

We gather statistical metrics by capturing critical transaction information at key interfaces of a design. A detailed discussion on the selection and reasoning behind the selection and capture of data can be found in another DVCon paper [2].

Figure 1 shows a simplified view of critical points in a system where transactional data can be captured. By knowing where and when the data was captured, and with knowledge of the transaction itself, we can determine how each IP interacted with any specific transaction, and look for patterns across the system.

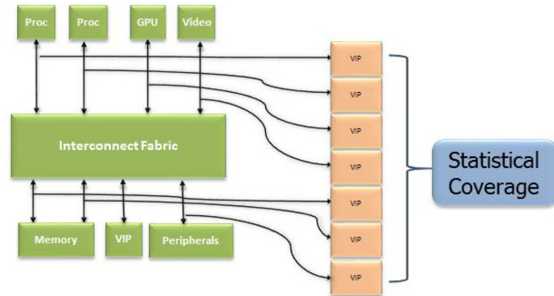


Figure 1: Distributed Data Capture

One example is on the main coherent fabric, we capture each processor transaction for analysis. We then correlate independent transactions that were captured at different sites, so that we can follow the progression of any one transaction across many points. Next, we search for interactions between transactions. Interactions occur where there is a a time overlap between transactions. This could be when multiple interactions need to share a resource, such as a bus. That allows us to examine arbitration, prioritization, or fairness, such as when interactions share an address. That allows us to look at ordering, coherence, and possible livelock issues. Finally, we can look for statistical patterns across a large set of overlapping transactions to draw conclusions about the operation of the device, the quality of the stimulus, or highlight areas of concern.

## 4. First results

One early result was to simply show transaction activity over the life of each test in a regression suite. Figure 2 shows an early example. This figure shows an immediate issue – the number of transactions falls off sharply over time. The stimulus environment had been running this way for quite some time; tests essentially stopped, but the simulation kept running for a long time after the test was done.

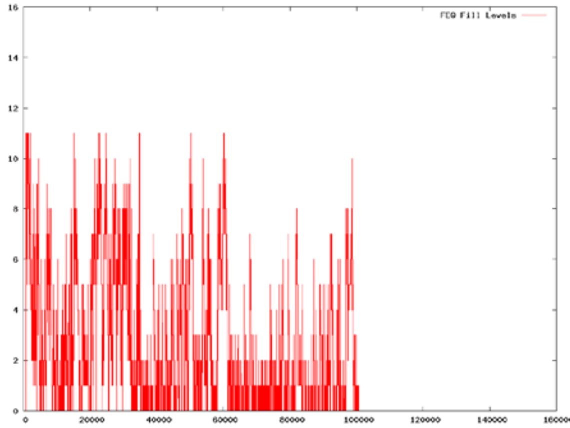


Figure 2 – Transaction density over time

As soon as this graph was available, a single trivial error in the constraints was identified that did not stop the test at the right time. This resulted in an immediate 60% improvement in regression CPU efficiency. The critical issue was not that there was a bug in the constraints – that was just a simple error. Without having the statistical coverage measurement, it was difficult to know that the bug existed: all tests ran and passed as expected, and any time that a bug was uncovered, engineers would examine simulations at the specific times when bugs occurred. Without higher abstraction-level metrics, there was no reason for anybody to look at the later part of a test. Even simulation profiling had not uncovered the issue.

This is an overall theme in our exploration of statistical coverage: If you can't measure it, you can't fix it. In some cases, the issue is simply being able to see data that is quite simple to understand. In other cases, it is a challenge to understand what you are seeing. In all cases, understanding what is happening is important to getting good verification results.

## 5. Statistical Coverage

As with functional coverage, there is a near-infinite set of possible measurements that could be made. Choosing statistical coverage areas must be both practical and actionable. Specifically, if a measurement does not provide information on how to improve the verification, then it is unlikely to be of much value.

Because any statistical coverage point can come from the correlation of any number of data capture points, the coverage plots can range from fairly straightforward runtime data, to highly abstracted system-level coverage. We will show examples across the range of abstraction.

One of the more surprising results for us was how much we could learn from straightforward count plots. When developing reasonably complex constrained-random stimulus, we have a conceptual understanding of what the distributions will look like. Our data showed us that what was actually happen was quite different than what we expected.

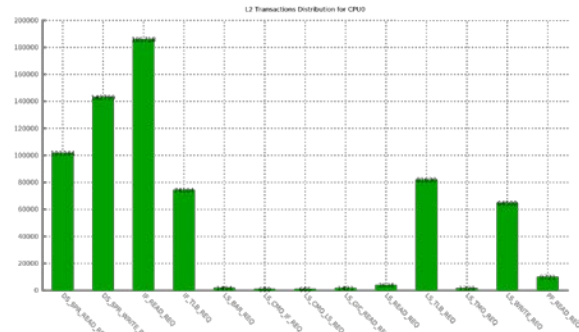

Figure 3 Cache operation distribution

Figure 3 shows a plot of the L2 cache operations across many tests within a specific suite. We had expected to see a fairly even distribution of operations on the L2. Since the stimulus is generated is at the processor, and not directly connected to the L2, there are a number of design-specific factors that come into play. Nonetheless, the L2 operation distribution was not an acceptable distribution. With the new statistical coverage, we could modify constraints to provide a better distribution. After some early constraint modifications, a rerun of the test suite resulted in the distribution shown in Figure 4.

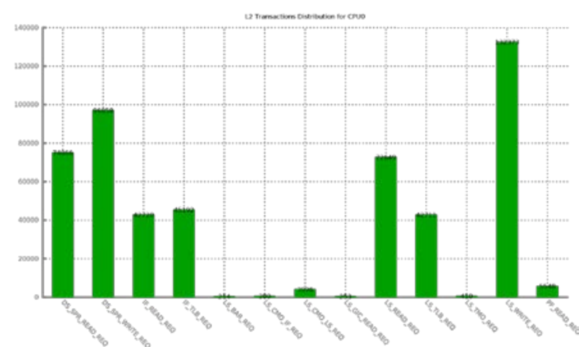

Figure 4 Cache operations distribution updated

We can see that we obtained significant improvements in the L2 operation distribution, although it is still not ideal.

## 6. Stimulus and Statistical Coverage

For the L2 cache operation distributions, it is pretty easy to understand what the targeted distributions should look like. One might be aiming at specific goals, in which case a skewed distribution is desired. Otherwise, a fairly even distribution might be desired.

In other cases, understanding the desired distribution may require some study of the domain, or the specific system architecture. In Figure 5, we show the first statistical coverage plot of the number of operations over a fixed time period.

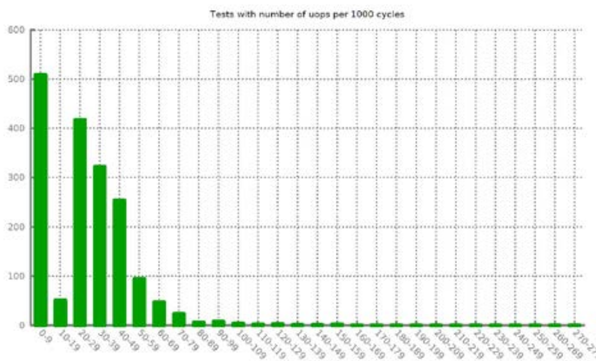


Figure 5: Operations per time slice

The constraints allow for a wide variety of results, as we would expect, but the plot shows that the distribution is non-optimal. Once the statistical coverage was known, it was possible to modify the constraints to provide a distribution that more closely matches the goals of the verification. Figure 6 shows the new distribution that was obtained.

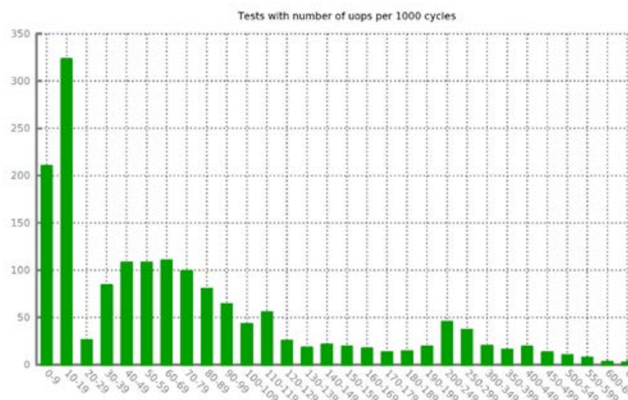


Figure 6: Operations per time slice updated

This figure also illustrates why understanding the distributions from statistical coverage can be complex. In larger environments, the constraints will interact with the busses, queues, caches, and other storage elements. Determining the required stimulus, expected and acceptable results, and potential improvements requires a good understanding of both system architecture and verification.

At times, this requires that the verification stimulus be highly unrealistic. One example is for cache coherence verification. Under normal operation, coherent cache interactions do not happen particularly frequently. That poses a challenge for coherence verification, where higher frequency results in a higher likelihood of uncovering a bug within a given number of cycles.

One commonly used approach to achieving higher frequency of cache coherence interactions is to lower the L2 read latency, while simultaneously generating unrealistic traffic. That can be effective in generating higher densities of snoop traffic. Figure 7 shows the first graph of L2 read latency.

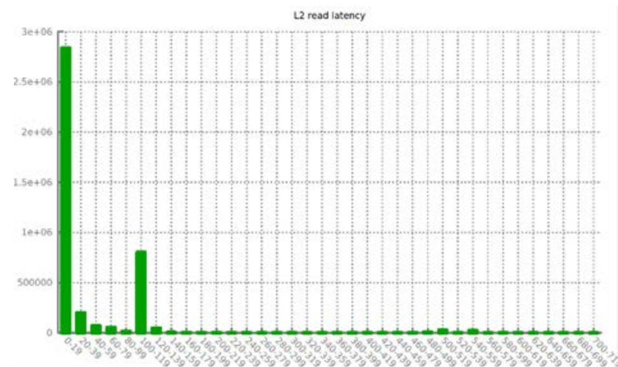


Figure 7: L2 read latency distribution

Without the information provided by statistical coverage, the read latency distribution was less than ideal. Once the distribution was measured and displayed, the constraints could be modified to allow for a still artificial curve, but with a better distribution over more realistic numbers as well. Figure 8 shows the result of the modified constraints,

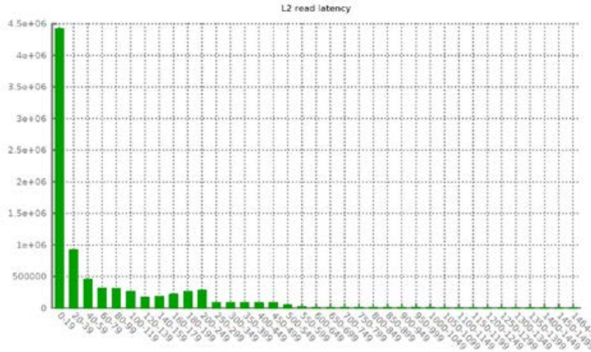


Figure 8: L2 read latency distribution updated

7. Analysis and Statistical Coverage

There is a category of system behaviors that are difficult to check with normal coverage results. Tjos is in cases where any individual operation works correctly, but the overall operation is not as expected. Examples of this include performance, Quality of Service (QoS), traffic shaping, and fairness. Correct operation is no longer about individual transactions. Rather, the ordering or modification of transactions based on other traffic becomes important.

Many existing solutions use counters and averaging to capture performance operations. That approach can provide clear, quantitative information about the system under test, but it may be difficult to use that information to explore corner cases, or determine the root cause of any particular result.

Here statistical coverage can provide detailed analytical results. As an example, Figure 9 shows a fairness plot for coherent bus transactions. Trends, such as transaction fairness are seen by plotting correlated data against time, source, or destination for example. This type of statistical analysis shows system behaviors that can only be seen over longer periods of time. In this particular plot, the overall result is close to expected, but one can also see the makeup of the data, and some outliers that may point to design issues, or may simply be due to the particular settings, data patterns or other circumstances.

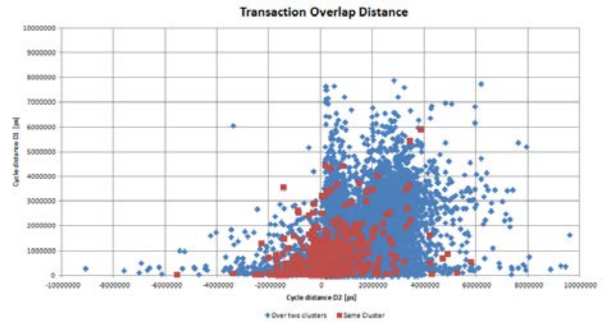


Figure 9: Coherent transaction overlap timing

Similarly, in Figure 10, the distribution of L1 queue fills show patterns of interactions and the limits of the constrained random stimulus. In this case, the range of cache queue fills are shown. Here one can see not only the upper and lower limits of fills reached via the stimulus, but also the distribution of fills, providing a few of cache use within this group of tests.

It should be noted that this type of graph is likely to be useful when separated out for specific types of operations or tests, rather than collecting large sets of data, where any outlier behavior could be hidden by larger sets of more typical behavior.

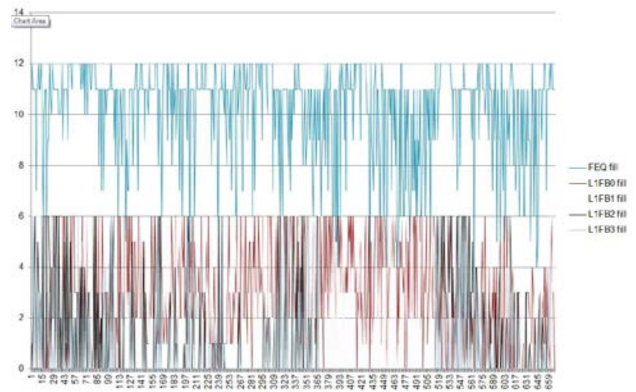

Figure 10: Distribution of L1 & L2 max fills

The ability to visualize the distribution of operations is valuable to understanding the limits of the stimulus, and the behavior of the system under test. Patterns are visible, outliers can be explored, and general correctness of system behavior can be examined.

While plots can be useful for understanding the system, being able to verify behavior in a regression environment is also important. Statistical coverage helps in two ways: first, plots can provide the understanding needed to determine what to check for

– what should be checked, and what are reasonable limits; second, the statistical coverage captures the data necessary to implement some of the checks, where data from multiple places must be correlated in order to determine the circumstances around particular timing or behavior.

## 8. Abstract statistical coverage

Up until now, we have been showing examples of statistical coverage where each individual transaction represents a data point, which is plotted either in time, or as a count against other data points. These are relatively easy to understand, and they can highlight areas that need attention.

There are also cases where statiscal coverage points are collected and correlated from many different points on many different tests for more abstract analysis.

It is possible to present a more abstract picture of how stimulus is interacting at a system level. Coherence cache tests are one example from our environment where understanding how the caches are interacting may require correlating processor operations, with cache state changes, with fabric transactions. A simple example of coherent cache interactions is show in Figure 11.

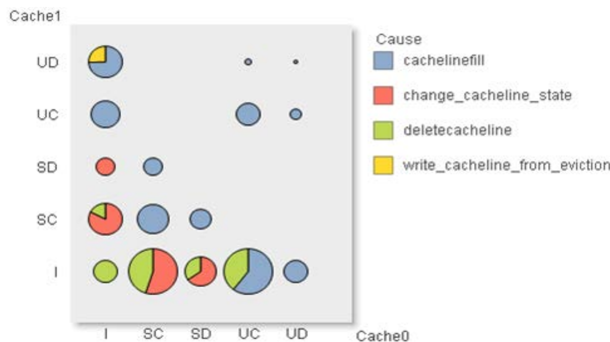


Figure 11: High level coherent cache coverage

This is a highly abstract plot of cache interactions that comes from a large number of individual data points across thousands of tests. This type of plot can be very effective in providing system-level domain information in a way that is reasonably easy to understand. This type of diagram would be exceedingly difficult to create from more traditional coverage methods, if it is possible at all. Here, we are looking at the overlap in time of data across the two L2 caches, and plotting when cache lines are shared between the caches, and the circumstance that caused them to be shared. Figure 11 shows a picture of cache interactions that look about like what one would expect. Most operations are within a single cache, but there are also a significant number of occurrences where data is shared across both L2 caches. One could easily conclude from this figure that the stimulus has reached a reasonable coverage goal for the L2 cache coherence.

What we have found is that abstraction can be very useful in displaying complex data relationships, but simply by the process of abstraction, it can also mask issues that may not be immediately clear. While the cache overlap diagram in Figure 11 looked reasonable, there are a number of more detailed plots that provided us with more information. We show one example of this in Figure 12, where we plot a cache state transition diagram. Here, the X axis shows the starting shared L2 cache coherent state – essentially the entire state diagram from Figure 12. The Y axis shows the ending shared L2 cache coherent state. From this plot, we can see one view of the distributed cache state transitions that have taken place.

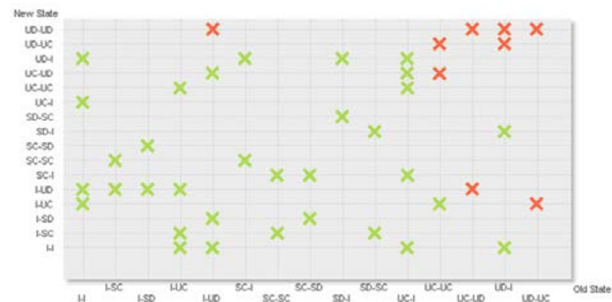


Figure 12: Detailed coherent cache overlap

While Figure 12 is still highly abstract, having been built from many thousands of data points, it does show more detail than the first graph. This statistical coverage plot highlights several key cache state transitions which need to occur in a complete test suite, but never occurred in the first set of stimulus.

The abstract plots are critical to understanding complex interactions that occur in system-level environments, for coverage, as shown in Figure 12, or performance, or general operation. Getting good information from any plot requires an understanding of both the system, and the limits of what any particular plot can show.

## 9. Human Understandable Statistical Coverage

One of the issues we faced in defining statistical coverage plots was to ensure that the plot was both useful and actionable. For a plot to be useful, it must be possible to understand not only what information the plot is providing, but also how the observed data relates to an expected outcome.

Quite a few plots can show useful information, but only after careful thought. There are many possible plots that clearly show interesting patterns, but it can be very difficult to determine what patterns one would like to see. Not having some reasonably clear insight into what is a “good” or “bad” pattern, along with some understanding of how to influence the data, renders the plot essentially meaningless. Even when the data presented is understandable, if it does not provide actionable information, then it has failed. The two most common reasons we have seen are: data that is too high-level, and data correlations used to create interesting plots, where it is unclear how, or even if, the correlations are relevant to the device under test.

## 10. Conclusion

We show how statistical coverage methods allow us to improve our understanding of a complex ARM processor environment, and to see where the constrained-random stimulus needs to be improved.

While existing code and functional coverage methods are critical to solving some issues, they are not sufficient. With the addition of statistical coverage, we are able to measure system-level IP interactions, and gain new insight into the performance of the simulation environment. That information lets us modify the stimulus, resulting in the uncovering of a number of functional bugs, and quantitatively improved verification performance and effectiveness.


[1] A. Meyer, H. Foster, “Metrics in SoC Verification: Not just for coverage anymore” DVCon 2013

[2] A. Efody, “Wiretap your SOC: Why scattering verification IPs throughout your design is a smart thing to” DVCon 2014

[3] M. Peryer, “Caching in on Analysis” Verification Horizons, October 2013, Vol 9, Issue 3