Understanding the effectiveness of your system-level SoC stimulus suite

Robert Fredieu, Mentor Graphcs, Wilsonville, OR, USA (bob*_fredieu@mentor.com*)

Alan Hunter, ARM Ltd, Austin, TX, USA (*alan.hunter@arm.com*)

Andreas Meyer, Mentor Graphcs, Wilsonville, OR, USA (*andy_meyer@mentor.com*)

***Abstract*— Today's verification engineers seeking a way to measure quality and completeness of a verification stimulus have access to a limited number of methods, usually involving a small handful of code and functional coverage tools. And while reviews of test plans and bug rates can also help in understanding verification quality, only the coverage tools provide a numerical measurement of stimulus quality that is direct, though often highly limited. Through in-depth domain-specific analysis of a system-level stimulus suite, this paper will examine the effectiveness of a few these traditional coverage tools. Our goals: understanding the strengths and limits of existing coverage metrics and providing insight into potential improvements.**

First we will explore under coverage – cases where there is functionality in the design that the stimulus suite didn't hit and so should have been reported as a coverage hole despite the tool declaring achievement of full coverage closure. We will examine the types of functional operations that were missed, why traditional methods were not sufficient, and possible approaches to provide improved coverage capabilities.

We also will consider over coverage – cases where the same stimulus was hit many times, indicating potential inefficiency of the verification suite. In such cases, we will describe areas where the regression efficiency is reduced due to excessive repetition of sequences, then explain how to identify these areas and modify the constraints accordingly. Ultimately, our aim is a paper that summarizes ways to improve both functional coverage and effectiveness by increasing the coverage of the system and reducing areas of over coverage.

I. Introduction

Understanding how a large SOC environment is reacting to stimulus is a challenging undertaking. As IP blocks within modern SOC's have more complex interactions with each other, understanding how stimulus interacts with each IP and how each IP interacts within the system becomes too complex for traditional analysis techniques.

In today's modern verification environments based on random stimulus generation, coverage methods are required to measure completeness of stimulus. These coverage metrics, mostly code and functional coverage, provide specific quantitative measurements about how the system reacted to stimulus. While they are both valuable, and have a long proven track record in helping understand the block level stimulus coverage, we examine areas where additional coverage information is needed for effective stimulus at the system level.

The traditional concern is of under coverage: cases where hardware designs have not been fully tested by the stimulus suite, yet traditional coverage methods indicate complete coverage. More effective coverage analysis would have reported a coverage hole. We examine a number of these within a complex SOC, along with the reasons for the under coverage report, and requirements for getting accurate coverage reporting.

A less common, but also significant concern, is of over coverage. While hitting a particular area many times is not likely to be of particular concern, if a stimulus suite is repeating the same pattern over and over despite randomization, then the effectiveness of the suite should be called into question. Neither code coverage nor functional coverage is likely to be effective in answering this type of question since larger patterns need to be examined in order to determine if the stimulus being repetitive at system-level.

To answer these questions, we use a combination of large data sets and domain-knowledgeable analysis to give us visibility into how the full SOC is reacting to stimulus suite. By examining and correlating data across the SOC, in conjunction with knowledge of protocols, we are able to gain a statistical coverage awareness of the system. We use that to measure the effectiveness of the stimulus suite, and contrast it with current coverage methods.

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.

II. 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 a DVCon 2014 paper [2].

As our analysis is connecting in to a complex existing system-level verification environment, it was critical to us that we didn't modify any part of the design or test-bench. Aside from being the only practical approach to connecting into an existing project, it also gives us confidence that our measurements are accurately portraying a real, working environment.

Figure 1 shows a simplified view of critical points in a system where transactional data must be captured. We are capturing large numbers of very small pieces of information throughout the system. 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.

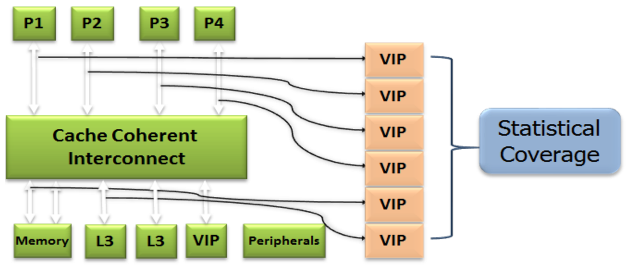

Figure 1: Distributed Data Capture

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 look for individual transactions that together make up a larger bundled transaction within the system. In complex protocols such as those found within arm SOC's, it frequently requires five or more transactions to make up a single bundle necessary to accomplish the original request. Examples of this include coherent memory operations, operations were data may already be stored within another IP block, and operations that need to be delayed or queued in order to maintain system-level performance requirements.

The need to look at bundled transactions in order to understand system-level intent is one case of where traditional coverage metrics begin to fall short. Determining which pieces are bundled with which other pieces is dynamic and distributed. Dynamic meaning that it is data such as tag ID that determines which transaction is correlated with which other transaction. Distributed in the sense that a correlated transaction may occur in a different part of the SOC than the original transaction, yet both components are part of the larger goal. Traditional coverage metrics are not well-suited for tracking dynamic or distributed operations to determine system-level stimulus coverage.

Next, we search for interactions between transactions. Interactions occur where there is a time overlap or other shared attribute 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 live-lock 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.

III. PROJECT OVERVIEW

Our metrics experiments are based on a modern multi-cluster high-performance coherent ARM V8 SoC project. The project uses a 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 all caches interacting with fabric to maintain coherence at the system level, tracking memory or packet transfers as multiple blocks compete for access, or ensuring that other system interactions such as interrupts don’t affect the outcome of specific transfers. Our goal with statistical coverage is to gain new insight into the inter-IP operations at subsystem or system level, and compare and contrast this insight with traditional coverage methods.

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.

IV. COVERAGE METHODS

Coverage is the classic concern in verification: is there hardware that hasn’t been tested? At the block level, this is likely to entail checks such as having both filled and emptied FIFO’s, or having covered all states in a state machine for example. These are also types of coverage that a combination of code and functional coverage are well suited to measure.

It should be noted that in earlier generations of SOC’s, individual IP blocks might share a common bus, but there would be very little interaction between the IP blocks. Communication consisted mostly of data transfers between two blocks, with all other components quiescent. The advantage of that type of environment was that there was very little additional functional verification needed at the SOC level since most of the inter-IP operations were relatively straightforward.

In many current SOC environments, such as our ARM multi-cluster SOC, performance and power optimizations require much more complex inter-IP interactions. One classic example is the set of transfers required for a coherent memory access. In this case, the requesting IP must issue a snoop operation to check which other caches contain the data of interest. The fabric, a different complex IP, must pass the snoop to relevant caches. The fabric may first use a snoop filter to determine exactly which caches need to be snooped, and which may be ignored. All other caches must react to the snoop and send the appropriate response back to the fabric, which in turn correlates the response to the original requester allowing the original operation to proceed.

The interesting points in this drawn out example are to show the complexity of distributed state machines that run across and between IP blocks, and the dynamic nature of determining what transactions belong to which state. Coupling this with the parallelism inherent in a modern SOC is where we expect to find differences between traditional and statistical coverage methods.

V. 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.

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 40% 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.

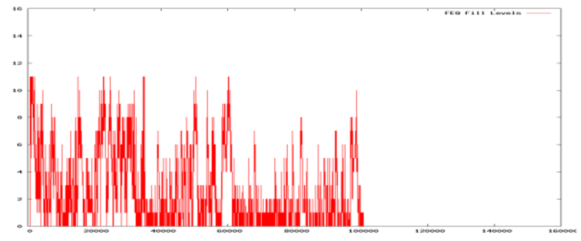

Figure 2 – Transaction density over time

This is an overall theme in our exploration of statistical coverage: If you can't measure it, you aren't aware of it. If you aren't aware of 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.

VI. 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.

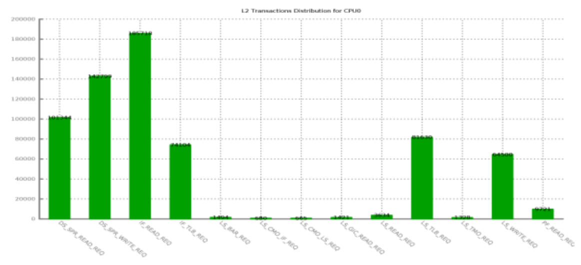

Figure 3 Cache operation distribution

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.

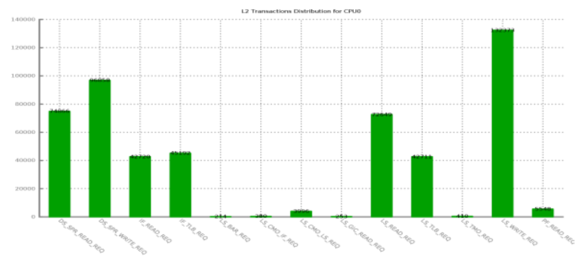

Figure 4 Cache operations distribution updated

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

VII. Under Coverage

While simple distributions can provide a comparison between expected and actual stimulus interactions, they do not provide a direct measure of system-level coverage. A distribution does not show a specific sequence of events that were or were not covered rather it can only hint at the effectiveness, by providing a comparison between what was intended and what actually happened.

To specifically quantify under coverage requires measurements of inter-IP operations. There are many examples of these such as QoS and QVN reordering, snoop filtering, or barrier operations, where a number of individual transactions interact causing the higher level goal to be attained. Note that many of these are architecture-dependent, similar to how functional coverage is frequently function dependent (some cases, such as FIFO operation could be argued to be automatic). We examine one case in detail: shared cache states. In this case a coherent cache line may be stored in one or more caches, following the domain specific rules. The transitions between cache lines is both dynamic and distributed: an n-way cache may put the line in any of the n locations and any caches in the system may be involved or not depending on the state of the cash tags. The coverage analysis must determine which caches are involved in the transaction, and how data is transferred.

When we analyze the regression for transactions we see a coverage pattern that is quite different from traditional methods. The diagram in figure 5 shows our first view of inter-cache transitions, including legal, covered, uncovered, and illegal transitions. Note that each data point within this plot requires the analysis of many, frequently 10 or more transactions, to determine that a single data point was hit. It also requires domain knowledge to determine legal versus illegal transitions.

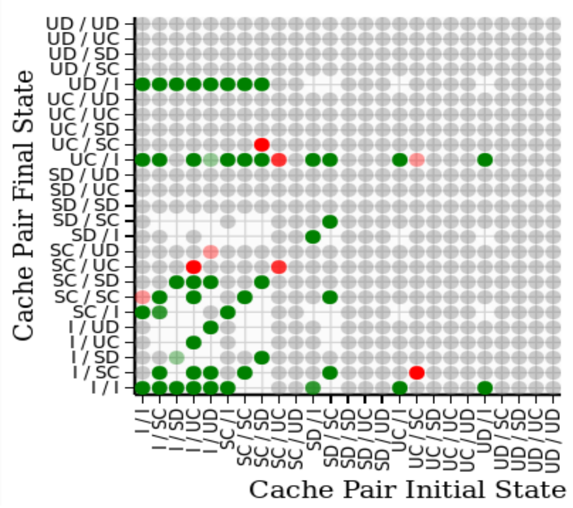

Figure 5: Cache operations distribution

Our first plot of this graph shows a number of places where the stimulus did not reach system-level coverage even though the relevant functional and code coverage indicated 100% completeness.

This system-level view illustrates a distributed state machine that has not been covered by the system-level regression. As with block-level coverage, some states are likely to be unreachable; even though legal, other components within the system are not able to generate the necessary stimulus to reach those states. This points to an interesting possibility for system-level dynamic reachability analysis. Other states are reachable and if uncovered, point to the need for additional stimulus. This example, showing coherent memory coverage is notorious for insufficient stimulus and difficult to find design issues.

Another aspect of system coverage is that architecturally illegal conditions may be reached. There are a number of potential reasons for this from hardware bugs to the more common illegal programming set up issues within the stimulus. One could view that as additional functional coverage, but where it is already difficult to hit legal, there is likely not much advantage in exploring illegal programming states. Just as an example, one common way to program states is through incorrect page tables setups, causing coherence failures among caches. In one case highlighted in figure 5, we ran into a setup issue in our stimulus that caused us to run stimulus in an illegal configuration.

VIII. Over Coverage

There are two issues to be addressed with over coverage. First, there is the loss of simulation efficiency. Testing the same thing many times is generally a waste of resources if there are no relevant changes in the system. Second, and more importantly, there is the likelihood that the stimulus was meant to be doing something else, but is instead doing the same thing over and over, which implies that some other intended coverage is being missed.

While one could argue that there cannot be too much testing, over coverage may well be the result of issues within constrained random stimulus. With a solid test plan, one would expect that stimulus generate coverage for each section or function in a design with the same reasonably even distribution-at least in the ideal case. If that isn't what is observed, then there is still the tricky issue of determining the difference between the intent and the observed activity.

One classic example of over stimulus due to constraints is in driving too much traffic into one interface, causing flooding. Once a component such as a FIFO or a bus saturates, traffic rates are controlled by saturated component rather than by the stimulus. At this point, the stimulus generator has lost control and traffic rates are being controlled by saturated component: any constraints specifying the rate of data are now ineffective, as the data rate is being controlled entirely by the saturated component.

More often, constraints generated by the stimulus are not behaving as expected in the system. Figure 6 shows the distribution with simple over coverage.

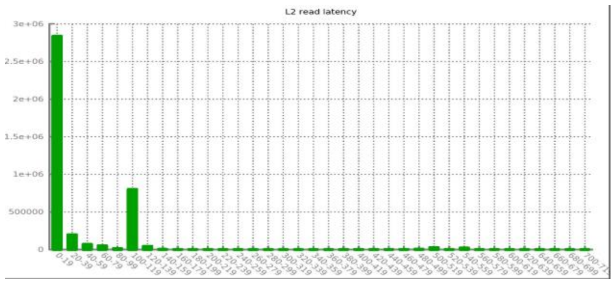

Figure 6: L2 read latency distribution

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 realistic 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 updated graph of L2 read latency.

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.

This example illustrates the complexity of system-level interactions, and the difficulty of predicting how a particular set of constraints will actually behave in a real system. The breadth of possible interactions is simply too large. More often, it takes considerable analysis to determine why constraints are not behaving as expected.

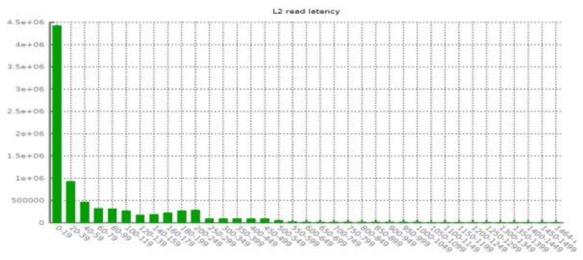

Figure 7: L2 read latency distribution

Without the ability to measure the actual over or under coverage, there is no way to know how well the constraints exercised the system, and thus no way to know where improvements are needed.

IX. Analysis and Statistical Coverage

There is another category of system behaviors that are difficult to check with normal coverage results. This 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 8 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.

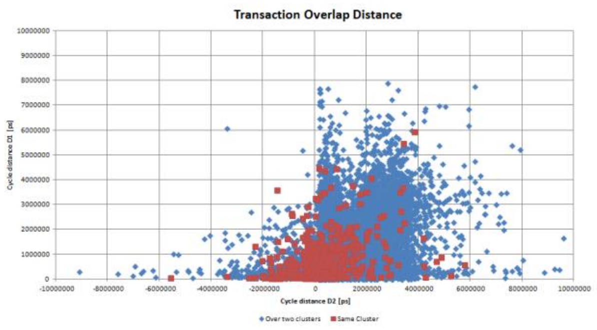

Figure 8: Coherent transaction overlap timing

Similarly, in Figure 9, 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.

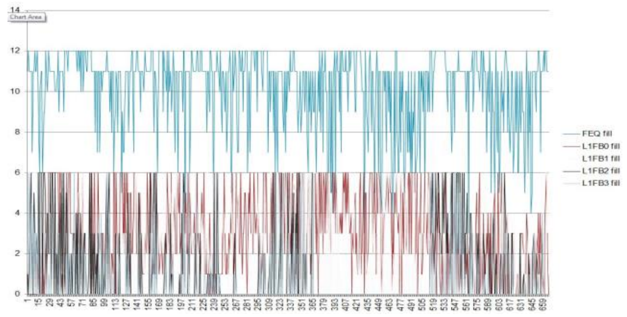

Figure 9: Distribution of L1 & L2 max fills

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.

X. 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.

The effort to connect an analysis tool is quite low. The time is mostly spent interpreting results, and determining the root cause of issues. However, once this is done, we are able to modify stimulus to hit whole new areas of the design that we code and functional coverage were not able to tell us about.

We saw a clear pattern where over-coverage indicated where constraints did not cause the expected reaction in the system. The cases we saw also had corresponding under-coverage, so the analysis was straightforward. It is conceivable that one could see only over-coverage if it causes under-coverage in another area that hasn't been analyzed.

Under-coverage comparisons are more straightforward, since the statistical coverage at system-level has similarities to functional coverage the block level. Once we saw how interactions between state machines actually occurred, we were able to add or correct stimulus for better coverage, which also resulted in the uncovering of RTL bugs.

References

[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.