2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Building Confidence in System level CPU Cache Coherency Verification for Complex SoC's through a Configurable, Flexible and Portable Testbench

Ruchi Misra, Shrinidhi Rao, Alok Kumar, Garima Srivastava,

Samsung Semiconductors India R&D, Bangalore,

ruchi.misra@samsung.com, shrinidhi.r@samsung.com, kumar.alok@samsung.com, s.garima@samsung.com

Youngsik Kim, Seonil Brian Choi,

Samsung Electronics, Korea,

ys31.kim@samsung.com, seonilb.choi@samsung.com

***Abstract*— The scope of CPU cache/coherency verification in an SoC context is covered mostly through memory accesses by the host CPU. Coherence protocols trigger complex interactions between heterogeneous mixes of caches and other system masters. Our regular SoC scenarios are less aggressive on cache functionalities as compared to IP-level testbench. Traditional approaches can sometimes miss bugs that arise only in a complex system level traffic. Writing large numbers of effective tests is difficult, time consuming, and error prone. Hence, we attempt to use a flexible and configurable testbench to enhance cache verification at System level. Through this, we also open up the possibility of at least 5X faster verification of other functionalities like Low power, IO Coherency, SLC/LLC etc.**

***Keywords*— CPU, Cache, Coherency, Simulation, SoC, Verification, Configurable, Testbench**

I. INTRODUCTION

A. The Problem Statement

Most SoC simulations rely on unit-level testbenches, especially in case of multi-core configurations since project timelines are short and simulation performance is greatly reduced in top level testbenches. Hence, we need to come up with a solution to generate many test attributes that verify CPU functionalities like caches effectively and quickly. Because of the presence of real Core RTL, CPU centric tests are generally coded in C and these manually created C tests lag behind the UVM in automation that has become mainstream now-a-days. The efforts of test creation, maintenance, reuse and leveraging for various projects is not addressed very well by these manually coded C tests.

There comes the need to automate the generation of test cases at SoC level to address the needs of diverse stakeholders and allow reuse across platforms and projects. The inherent requirement to enhance our confidence in CPU design along with test portability, reuse, and a short turn-around time led us to explore a configurable and flexible stimulus environment to achieve our goals.

B. The Motivation

In VLSI industry, the IP providers generally run an exhaustive verification of the design. The end user does not verify the internal functioning of the IP since it is certified to work by the vendor. When such an IP is integrated into a complex SoC, the connection of the IP with other system components could affect its functioning efficiently. Thus, there is a requirement to verify these verified components again in the SoC context. The block in question for us is the L1-L2-L3 cache system within the ARM processors.

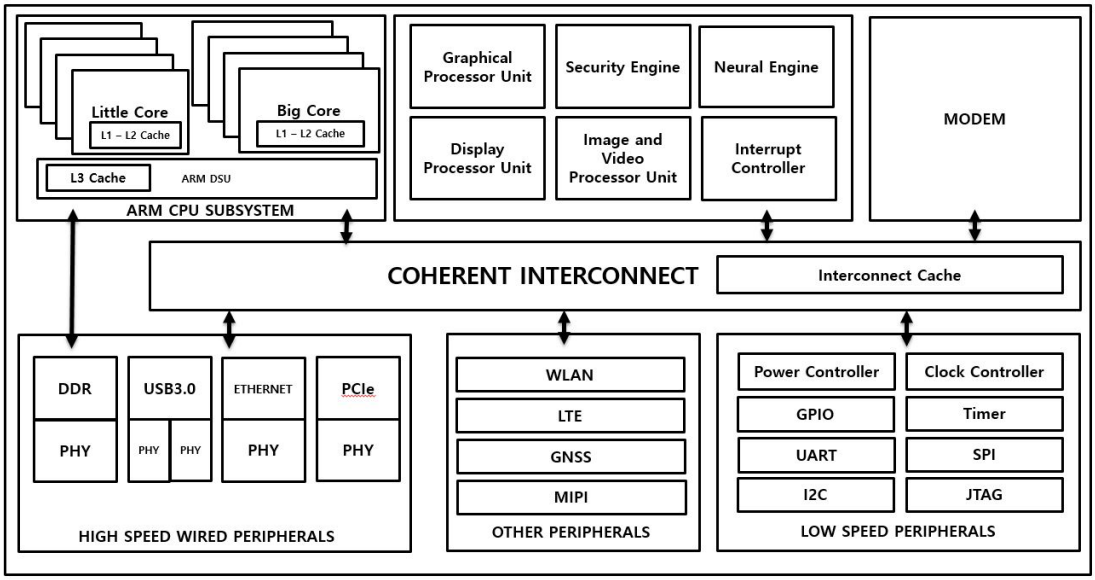

Figure 1. System On Chip architecture with a multi-core system

As depicted in Fig 1, In modern heterogeneous System-on-Chip (SoC), the cache hierarchy is one of the most complex and problematic components. Because of a huge number of possible cache hierarchy states, verification of the cache hierarchy requires numerous complex tests, which becomes the main problem for functional verification. The most common solution to this problem is to develop specialized test generators for a single level of cache. However, for the entire cache hierarchy, such generators cover only a few localized subsets of the global state space leaving large gaps between these subsets.

In a repetitive and progressive development of configurable IP core environment, Regression testing in design verification plays a vital role. As size of the design continues to grow accordingly increases the complexity of verification and regression time. However, it is a very well-known fact that maintaining regression quality and reducing regression time is very difficult specially in an SoC environment containing such IPs.

II. Scope of Configurable and Automated Stimulus in SOC CPU Design Verification

Shelly Henry et al. [1] through experiment with inFact by Mentor Graphics were able to conclude that using Portable Stimulus gave a significant advantage on time required for coverage closure as compared to traditional UVM constrained random methodology. It becomes even more relevant when the testbench implements complex cross-coverage where it becomes difficult to add distribution constraints to achieve coverage. Larry Melling of Cadence described the need for a Portable Stimulus standard as follows: “Portability of reusable test cases has long been a goal for semiconductor verification and validation teams. No one wants to ‘reinvent the wheel’ by having to rewrite similar tests again and again” [2].

The focus is on the areas related to cache and coherency verification that were not exercised exhaustively at SoC verification. Verification of the CPU design in our multiple complex mobile/automotive or wearable SoCs is mainly done through C tests. Using a configurable and automated test environment enables us to focus on targeted feature test generation, saving time and effort. This configurable testbench takes the specific configurations of the SoC as user-defined inputs, generates generic C code for some standard operations, and combines them to generate final C scenarios that run on embedded CPUs and exercises the system through diverse relevant solutions. The code generated for the simulation platform can then be re-used by the other

dependent environments, allowing reuse across various verification platforms. This methodology is aimed to target multi-core I/O and cache coherency, resource-aware activity coordination, stress testing, performance testing, and tuning, and low power attributes. We have explored the feasibility of adapting Portable Stimulus Standard in a C based test bench, focusing on coverage metrics, time to generate test cases, flexibility and re-use across platforms.

III. PORTABLE TEST AND STIMULUS: AN ACCELLERA STANDARD

Portable Test and Stimulus Standard is a new Accellera standard for an abstract definition of the verification intent that can be used for stimuli generation for different types of verification environments and at different levels of design hierarchy.

In functional verification, several different languages and techniques are used to generate verification stimulus files depending on whether a block, subsystem, SoC, or system is being verified. When verifying an RTL block and subsystem, System Verilog is frequently used although "e," SystemC, and VHDL are also used. At the SoC and system level, embedded software is frequently used to exercise the design. Several challenges result from different languages and techniques being used for block- and subsystem-level verification. It is difficult to leverage block- or subsystem-level test scenarios at the SoC and system level. In addition, the embedded software that drives stimulus in SoC and system level environments does not provide support for automated stimulus generation the way that languages such as System Verilog do (e.g., constrained random generation) in block- and subsystem-level environments.

Currently, there is no single standard way to specify intent and behavior that are reusable across target platforms (e.g., emulation, silicon, simulation, etc.). The Portable Stimulus Working Group will create a standard around enabling verification stimulus to be captured in such a manner that enables automation of stimulus generation and allows the same specification to be reused in multiple verification languages.

Even though the idea behind the standard is clear and well received generally, there is still some time to make it work seamlessly inside of the interpretation tools. In this paper, we take a similar inspiration and focus on vertical reuse of portable models which is basically about testing block-level designs at the subsystem or system-level.

C. Related Work

As complex SoCs get designed, many of the design components get reused from Subsystem level. This opens up an option to reuse verification environment as well, allowing sharing of verification efforts. Although UVM provides excellent reusability, it cannot be reused seamlessly across verification platforms like emulators. Matthew Balance [3] laid out reuse strategies to be followed while building a test bench with PSS. After identifying primary target and asset to reuse, one must design test, libraries, checkers keeping in mind reusability and tradeoffs that come with each axis of reuse. He establishes that the vertical re-use between subsystem and SoC context requires the highest amount of effort, but provides maximum benefit while technique reuse between simulation and prototype provides least benefits. He further establishes strategies for result checking and identifying target behaviors of the system [4]. Efrat Shneydor Et Al. [5], and Revati Et Al. [6], address the underlying infrastructure and test generation flow of the Portable Stimulus Standard Engines. Portable stimulus Standard has since then proved to be a very capable methodology and has found many researchers working on it to enhance the performance [7]-[10] and efficiency of verification.

Design Verification challenges lie in quality of IPs, interconnections, quality of verification vectors, runtime, timeline, tools and methodologies. These challenges increase with design complexity. In the context of a complex SoC with multiple PCIe IPs with dual mode support, varied speed support and link width variations, it is hard to create virtual sequences for concurrently accessing or synchronizing multiple tasks. Addition of a controller to this already complex SoC, would require multiple test bench files to be modified.

Thanu Et. Al [11] took up the challenge of accelerating the verification cycle and improve the quality of verification using Portable Stimulus Standard over configurable System Verilog-UVM testbench methodology. They were able to create a solution with a Portable test and Stimulus Standard tool in generating many test scenarios for a coreless SoC environment in PCIe's concurrent configurable testbench. This methodology

helped in automating an entire process of manually developing complex system level coverage driven tests to verify PCIe subsystems in a System on Chip. Initial one-time effort required was more requiring almost double the time compared to manual creation of test bench. They concluded that subsequent projects would require much lesser effort, since generation of test cases are now faster. The created test cases were more complex, but debugging would be easier owing to self-checkers and more coverage due to constrained random scenario. Since the tests could be run concurrently for both controllers, runtime and disk space consumption was smaller.

IV. METHODOLOGY

D. Test Bench Modification

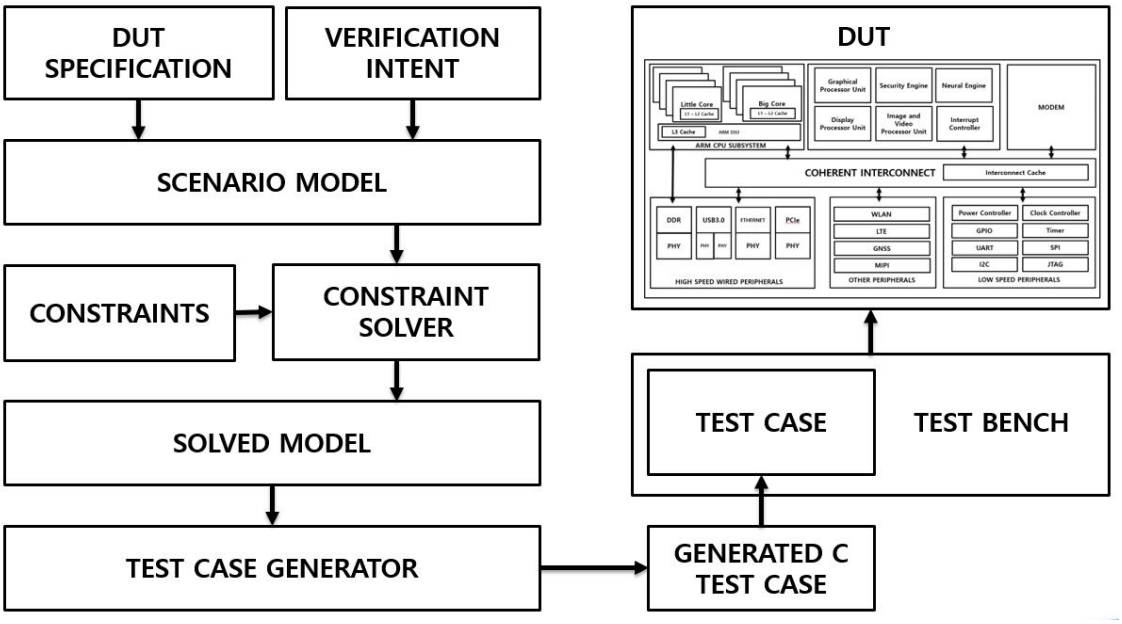

Figure 2. Flow chart depicting generation of test case and usage

As shown in Figure 2, Test cases to verify the CPU subsystem are written in C as per the DUT Spec and Verification Intent. This process would involve using pre-built modules, which can perform specific tasks to generate test code. While generating the c code, constraints set by design and user are considered. The final generated C code is then plugged in to the test bench and compiled using the ARM gcc compiler to generate hex code. This hex code is loaded into memory from which the CPU starts execution upon reset release. This C-based test bench is supported by UVM sequences to initialize and configure signals and systems before the chip boots up. The simulation is then run on the Xcelium simulator provided by Cadence.

The configurable test-bench requires an input file with details of the system, such as the number of cores and the type of each core, number of caches and their respective cache sizes, cache allocation policy, information on the MMU table, allowed power modes etc. The tool uses this information to generate small test case modules or functions, which are individual tasks that perform one atomic task. For example, by using cache size information, the tool would generate a task to perform "n" read or write accesses such that it fills up the L1 cache. We can then combine these smaller functions to create complex scenarios as per our requirement.

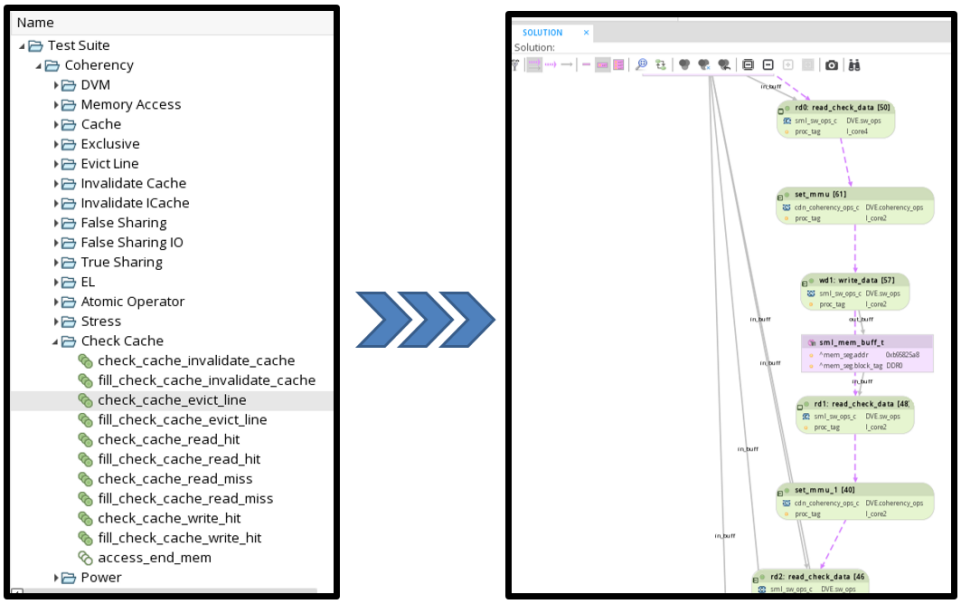

Figure 3. Test Solution generator

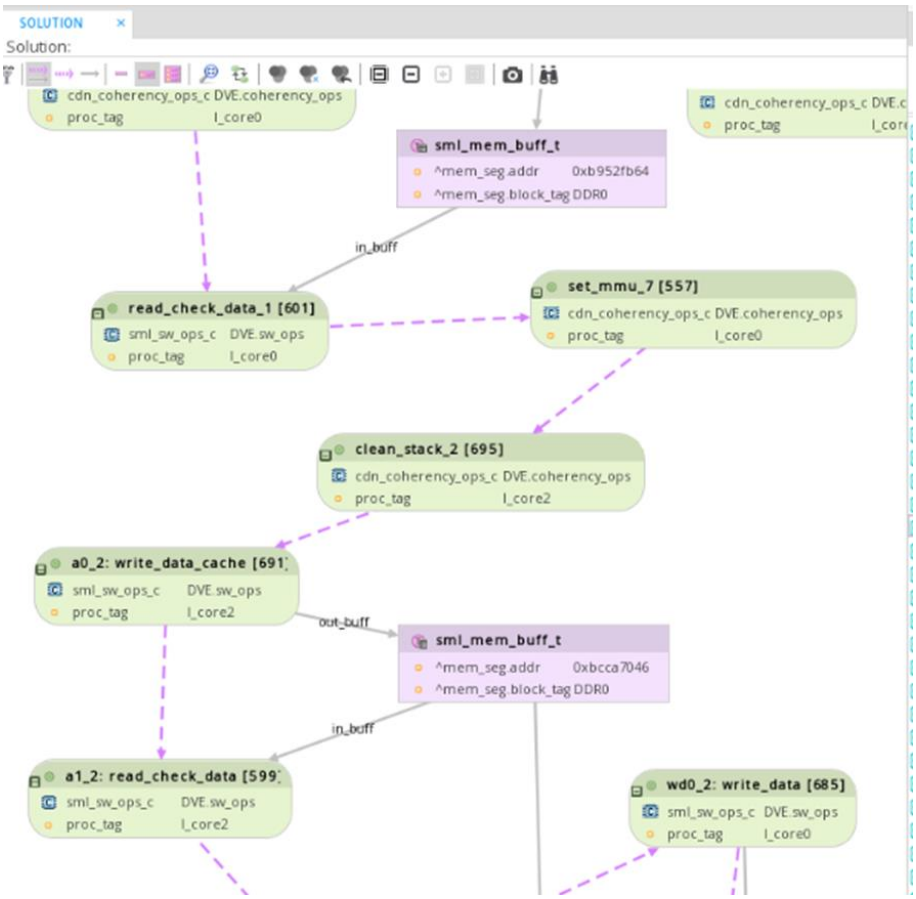

Figure 4. Interconnect of simple actions leading to a complex scenario

Figure 3 shows an example where the tool has generated a probable solution to test eviction of cache data. Solution generated can be analyzed in depth and we see in Figure 4 about how complex the solution generated for this task is. Each action can further be updated or constrained according to the verification intent.

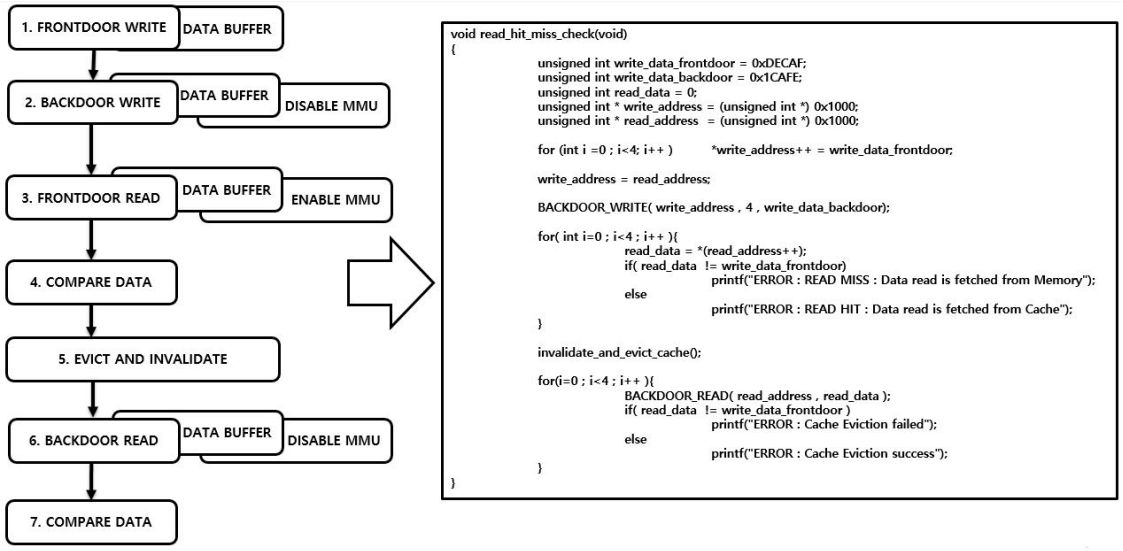

Figure 5. Mapping of test sequence to generated c code

On the similar lines of Figure 5, which shows a flow chart depicting the basic functions used in the scenario, constraints set, and the features are targeted. We were also able to generate a coverage report based on the flow chart, which allows us to modify, add, or remove the function modules.

The next step would be to generate the actual C code from the flow chart. The tool generates a C file with all the necessary functions containing the test sequences and checkers, and a main function which contains all these functions. A standard test case template was then created, which contains the system initialization and boot flow, after which the verification functions start. This function is marked as a dummy task.

The engineer now generates a directed test case using this and uses the standard template to call this function after boot flow by replacing the dummy function with the generated function. After including all the necessary dependencies in the test case, the C code is compiled and loaded into memory for the CPU cores to read. Thus, this configurable test generator gave us a simple plug-and-play system where the manual work involved was limited to replacing the dummy function in the template with the function name generated by the automated test generator.

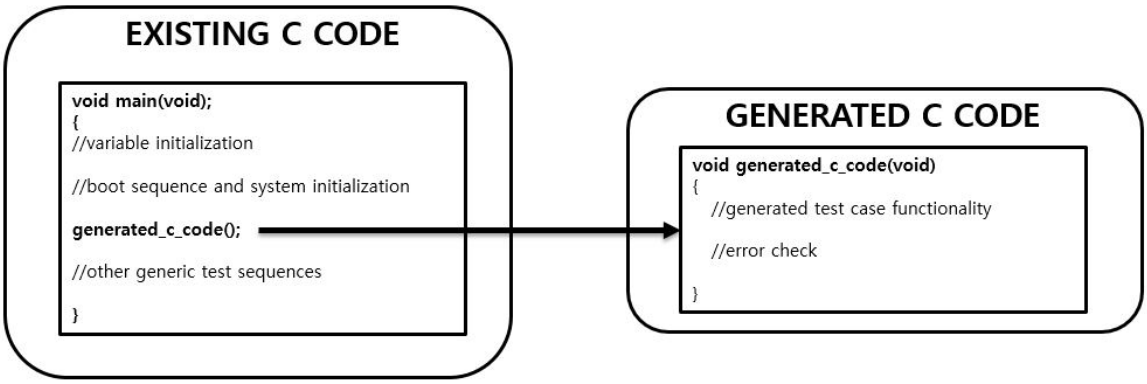

Figure 6. Plug and play usage of generated C code

In the Fig 6, the above explained step is depicted through a code snippet. On the left side we can see the existing test code in C which calls the automatically generated C function on the right that can be easily plugged in. The effort involved in creating the test case is large in PSS based test bench since the probability of catching initial issues is large. Once the model is tested and well established for one SoC, the test case generator can be used across projects that involve similar architecture or involve only modification of specifications. Unless there is a major change in chip architecture, the Portable Stimulus standard can be easily deployed across projects. Figure 7 shows a complete view of the tool, with scenario and solution generators docked in one screen.

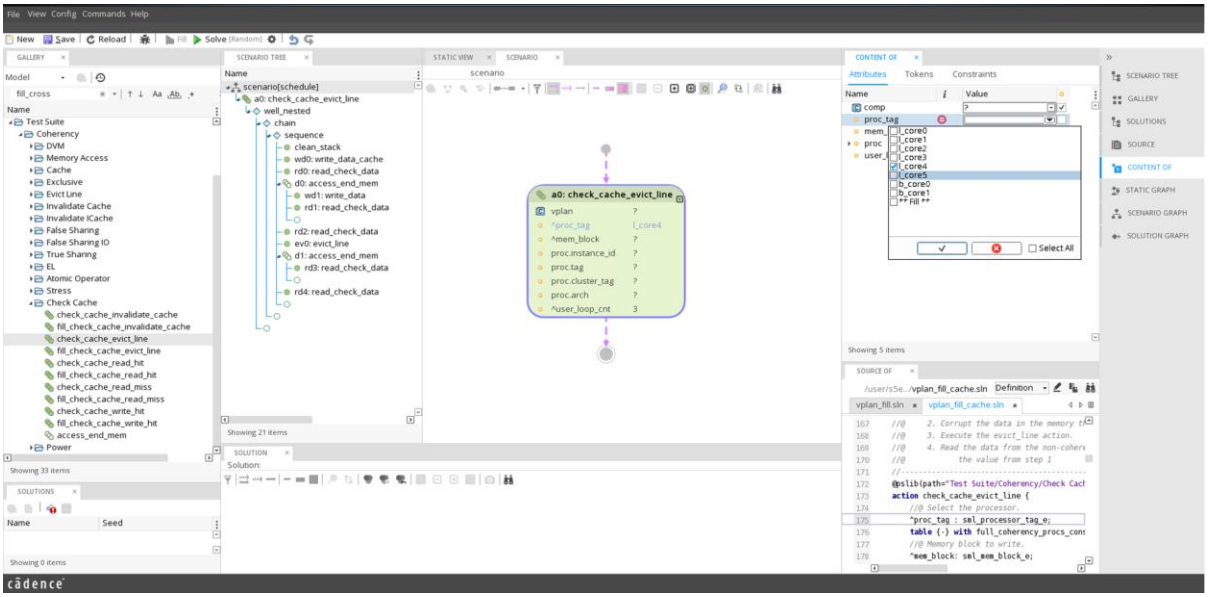

Figure 7. Complete tool view

V. SOME PRELIMINARY CASE-STUDIES

The pilot program to conduct some case studies was run on an Exynos Mobile SoC environment at Samsung India R&D Center, Bangalore. We mainly focused on generating test cases for verification of cache features since that was our primary target. We were able to generate coverage reports on the types of cache actions, which helped us to generate multiple scenarios for complete verification. We added directed test cases for cache allocation, cache boundary checks, read hit and miss, write hit and miss, cache eviction, cache cleaning and invalidation, false and true sharing, and multiple stress scenarios involving the same. In very short span of time, we could run a good number of test cases that were generated through this configurable test creator

- Fill Cache Boundaries: A stand-alone Allocate to Cache task allocates a memory block to cache. And another multiple Cache Access task writes to cache regions. Now extracting the cache size information, this compound task created an access of the memory chunk which is equal to the cache size. Using single tasks, create cache eviction, overflow and clean and invalidation operation of each of the core caches.
- Cache State Transitions: Using the different variety of opcodes supported, this complex test covers various cache state transitions with the help of an in-built checker. Such checks on state transitions help in building confidence on the design and stimulus. It also helps to verify read and write hit/miss scenarios in CPU Caches.

- Data Sharing Scenarios: True Sharing and False sharing of data between cores in a coherent setup can significantly degrade cache performance, in systems when smallest data size accessed is smaller than cache line size. Testing such patterns of data sharing can give designers an insight into cache performance and performance enhancement mechanisms.

VI. RESULTS

There is a one-time effort involved for any verification engineer to integrate the automatic test generator into the SoC environment. Initial setup needed multiple iterations to be run since the generated code was not very compatible with the compiler and the system it was run on. This effort is minimal in existing TB since they are pre-verified and well established from legacy projects. Once testbench is ready, new test case creation is faster in the new approach. It took us about 2 months to get the initial setup ready and about 3 months to run the complete set of tests and verify the attributes. We estimate that the same work would take an engineer about 400 days if done manually. This number is estimated taking into consideration the number of lines of code dumped by the test generator, code complexity and iterations to fix test bench issues. We found multiple compatibility issues between tool version, simulator and compiler that caused few tool bugs including three major bugs which caused manual waveform review of generate C tests and their simulations.

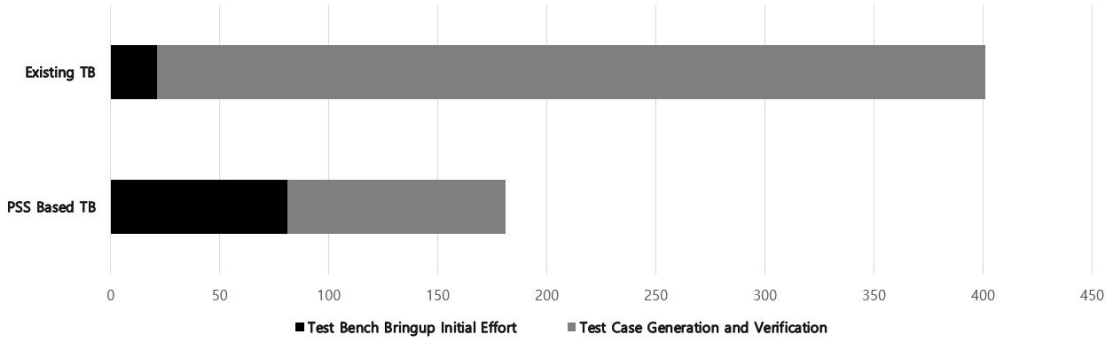

Figure 5. Comparison of effort involved between manual and test case generation

E. Advantage of Using Automated Test Generator

- The initial effort to setup the tool, create models and generate test cases is considerable. But once established, the effort required for test case creation for all future mobile SoCs are minimal.
- We were able to generate coverage reports after creating all the scenarios as well as generate the report at each step of test case creation. This gave us an idea of verification holes and the test cases that had to be added.
- More coverage due to constrained random scenarios and the flexibility to modify tests as per requirement at each step. The easy test case generation mechanism makes filling the verification holes faster and effective.

VII. CONCLUSION

The work represented in this paper uses a specification to create a single representation of stimulus used across many levels of integration. Correct-by-construction generation of complex concurrent multi-core and multi-thread tests is possible through this configurable stimulus. The ease of portability of test intent and test suite to various projects makes it an attractive methodology for verification. It helps to verify more corner cases with its automatic test case creation capabilities. The platform is portable, re-usable and scenarios can be easily re-produced. The number of test cases that can be generated is significantly higher compared to manual test development. It helps the user address areas that need focus from Design Verification in the context of SoC CPU Cache and Coherency Verification. It also provides a measure of completeness through the coverage reports of functionality, flows, and dependencies. We suggest users of this standard follow the similar approach of creating targeted test functions which are flexible according to specification set and then plug the code into

the existing test bench. This would significantly reduce the development time needed since only a small portion of the code would need to be updated across projects.

VIII. ACKNOWLEDGEMENT

We thank Cadence India for their extensive support with their tool Perspec, allowing us to experiment on the Portable Stimulus Standard.

IX. FUTURE POTENTIAL

- Since the pilot program has given satisfactory results, it is recommended to implement a portable and configurable testbench in all subsequent complex mobile and automotive SoCs
- Though the sample size of the cache verification tests is not huge, there is an opportunity to tune our models to improve the scenarios generated for stressing the cache operations.
- The coverage reports generated can also be improved to have more meaningful bins
- Our pilot program was targeted to the caches contained in the CPU subsystem, and we would like to extend the functionality to include the Last Level Caches present in the interconnect and caches present in other processors around the SoC, allowing the verification of I/O Coherency as well.
- With newer Exynos chips being based on latest ARM architecture, the library needs to be enhanced to include new architecture-based models to target verification of newer cores and features bundled with the newer ARM core micro-architecture.
- Another important feature that we need to target is Low Power verification, which generally consumes considerable time and effort. Test suite to target power and clock control would be more effective.
- By combining power and cache operations, confidence in the Silicon can be greatly increased and with the help of re-usable testbench, the process can be achieved within considerably short span of time.

REFERENCES

[1] S. Henry and N. Regmi, "How to Close Coverage 10x Faster using Portable Stimulus Standard - A Case Study," 2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV), 2018, pp. 28-30, doi: 10.1109/MTV.2018.00015.

[2] G. Moretti, "Accellera's Support for ESL Verification and Stimulus Reuse," in IEEE Design \& Test, vol. 34, no. 4, pp. 69-75, Aug. 2017, doi: 10.1109/MDAT.2016.2642898.

[3] Matthew Balance, "Unleashing Portable Stimulus Productivity with a PSS Reuse Strategy", presented at DVCON US 2019

[4] Tom Fitzpatrick, Matthew Balance, "Results Checking Strategies with Portable Stimulus", presented at DVCON US 2019

[5] Revati Bothe Et Al., "Generic testbench/PortableStimulus/Promotability", presented at DVCON Europe 2019

[6] Efrat Shneydor Et Al., "Portable Stimuli over UVM using portable stimuli in HW verification flow", presented at DVCON Europe 2019

[7] Theta Yang Et Al., "Bridge the Portable Test and Stimulus to UVM Simulation Environment", presented at DVCON US 2018.

[8] Karandeep Singh Et Al., "A pragmatic approach leveraging portable stimulus from subsystem to SoC level and SoC emulation", Presented at DVCON India 2019.

[9] Niyaz K. Zubair Et Al., "Increasing Regression Efficiency with Portable Stimulus", Presented at DVCON US 2019.

[10] P. Bardonek and M. Zachariášová, "Using Control Logic Drivers for Automated Generation of System-level Portable Models," 2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2020, pp. 1-4, doi: 10.1109/DDECS50862.2020.9095708.

[11] Thanu Ganapathy Et Al., "Acceleration of Coreless SoC Design-Verification Using PSS on Configurable Testbench in Multi-Link PCIe Subsystems", Paper ID 7069, presented at the DVCON 2021, Applications of the Accellera Portable Stimulus Standard Track, Mar 5,2021.