Practical Scheme to Enhance Verification Turn-Around-Time by Using Reusable Harness Interface (RHI)

Jongpil Jung, Hyunju Lee, Jaejin Ha, and Yonghee Im[1]

[1]SK Hynix, SoC IP&DV, Seongnam, Korea
({jongpil.jung, hyunju5.lee, jaejin.ha, yonghee.im}@sk.com)

***Abstract*- System Verilog (SV) and UVM methodology substantially get settled as a common Design Verification (DV) methodology in modern SoC design. One big drawback from this DV methodology is turn-around-time (TAT) taken to testbench build-up, scenario development, and debugging. TAT is getting enormous as the size of SoC is bigger and DV engineer should tough it out during the verification phase.**

In order to solve the TAT issue out, we propose practical schemes that admiringly give us a drastic cutback in TAT. The first scheme this paper presents is combining conventional DPI-C based testbench and UVM based one. Several good schemes regarding merged DPI-C and UVM have been proposed in the past, but we proposed practical way to make the merged testbench by harness interface. On top of that, we propose Reusable Harness Interface (RHI) to reduce scenario development TAT and debugging TAT by reusing the same test codes in different hierarchy - such as IP, Sub-system, and TOP level DUT.

***Keywords*- Design Verification, System Verilog, UVM, SV-DPI, TAT, RHI**

I. INTRODUCTION

SoC design only requires more and more resources – both time and human – as its chip size keeps growing in terms of logical gate number. What's worse for engineers, project schedule becomes shorter and shorter due to fierce competition in market. Considering *design verification* (DV) takes up more than half of the total design time, such trend is concerning to DV engineers especially. There are already many efforts put into reducing DV *turn-around-time* (TAT) in recent researches to catch up this trend.

DV TAT can be categorized into three kinds: *testbench build-up TAT*, *scenario development TAT*, and *debugging TAT*. Among these three kinds of TAT, this paper presents a practical scheme to dramatically reduce scenario development TAT by combining conventional DPI-C based testbench and UVM based one. On the top of this merged testbench, the introduction of *Reusable Harness Interface* (RHI) helps reduce debugging TAT by reusing the same test codes across the hierarchy such as IP, Sub-system, and Top level.

The remainder of this paper is organized as follows. Verification TAT issues along with motivation of the proposed schemes is briefly introduced in Section II. A scheme to merge DPI-C based testbench and UVM based one is presented and RHI is proposed in Section III. In Section IV, the scenario development TAT gain of the proposed schemes is shown in experimental results. Finally, conclusions are given in Section V.

II. VERIFICATION TURN-AROUND-TIME ISSUES

One research found out that 22%, 22%, and 39% of the total verification time are spent on testbench development, creating test and running, and debug respectively [1]. In other words, the total amount of time for above is overwhelming superiority across whole DV time. During the time, most of DV engineer frequently compile and run testbench over and over, and spend their DV time on these TATs with unpredictable iterations. What is worse, as the size of design is bigger and the complexity of design is growing, more TATs are enforced. Consequently, these TATs are intimately affect the number DV resources and project time.

The verification time categorized aforementioned 22%, 22% and 39% is tightly connected with testbench build-up TAT, scenario development TAT, and debugging TAT respectively. In this paper, we attend to scenario development TAT and debugging TAT which can influence on 61% of the total verification time with our proposed schemes

A. *Conventional DPI-C based Testbench*

In this paper, conventional DPI-C based testbench refers to the one coded in Verilog where designers can run directed testcases by mimicking bus commands as if they come from CPU. This approach had been favored by designers because test codes of the testcases can be easily reused by firmware (FW) engineers, as long as underlying infrastructure such as hardware abstraction layer (HAL) and low level driver (LLD) is implanted into FW platforms. Figure 1 show the conceptual architecture of test FW in C domain which shows HAL and LLD at driver layer. Another big advantage is relatively shorter total compile time during simulation. Since the test codes are mostly written in C language, they can be compiled separately from testbench. Later, compiled test codes are dynamically linked to the testbench before a simulation run. This separate compile capability can benefit us shorter TAT while developing and updating test scenarios.

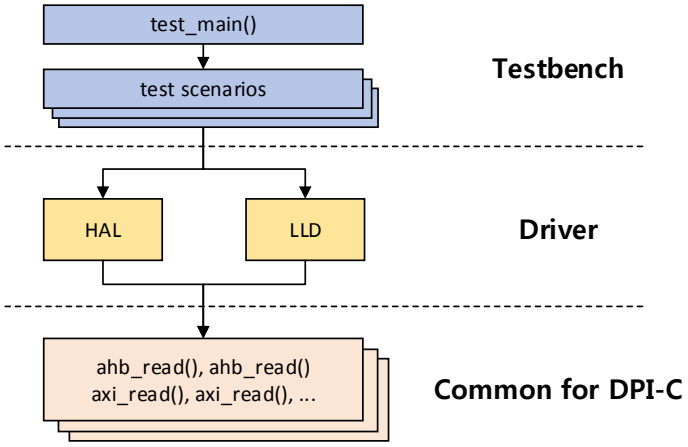

Figure 1. Architecture of Test FW in C domain

B. *UVM based Testbench*

UVM have become mainstream in DV world. Considering development efficiency, reusability of verification environments and verification IP (VIP), DV works with UVM are absolutely imperative. Recent research shows that UVM adoption has increased by over 70% in 2017 [1]. One of drawbacks of UVM is irritating re-compiling TAT while developing scenarios.

In a typical UVM based testbench, a test scenario is a collection of UVM sequences. Creating UVM sequences is not an easy task. It should rather be a painful and enduring "trial and error" process. Thus, UVM sequences development demands a significant amount of TAT. TAT only gets longer as the scope – the sequences are applied to – becomes bigger: e.g., from IP level to sub-system level. What's worse, you may find yourself re-writing a new sequence to test the same feature in a different hierarchical level, thereby aggravating scenario development TAT.

There were already some attempts to combine the above two testbenches, but they are only focused on how to reuse C-based test sequences [2-5]. Instead we are focused on how to minimize TAT in hierarchical verification from the merge. The first idea is to efficiently merge C-based test sequences into UVM environment in order to reuse already existing test codes (*scenario development TAT*). The second idea is to make harness interface reusable across the hierarchy not only to reuse test codes, but also to easily reproduce high level failures at lower level environment, thereby alleviating debugging efforts (*debugging TAT*). Followings are these two key concepts.

III. PROPOSED SCHEMES

A. *Merge DPI-C Sequences into UVM Environment – Step 1*

Unlike other previous attempts, we used harness interface as intermediary to merge DPI-C and UVM as shown in Figure 2. Harness interface is often used to contain virtual interface to bridge DUT and testbench in typical UVM practices. On top of such role, we added to the harness interface a new role of linking DPI-C and testbench. Harness interface imports and exports DPI-C functions and C based tests call those functions. DPI-C functions are translated into UVM sequences inside the harness and the UVM sequences are fed to the sequencer in UVM testbench as

denoted in red dotted arrow line in Figure 2. Blue dotted arrow line indicates normal UVM sequences and they are multiplexed with the C based test sequences.

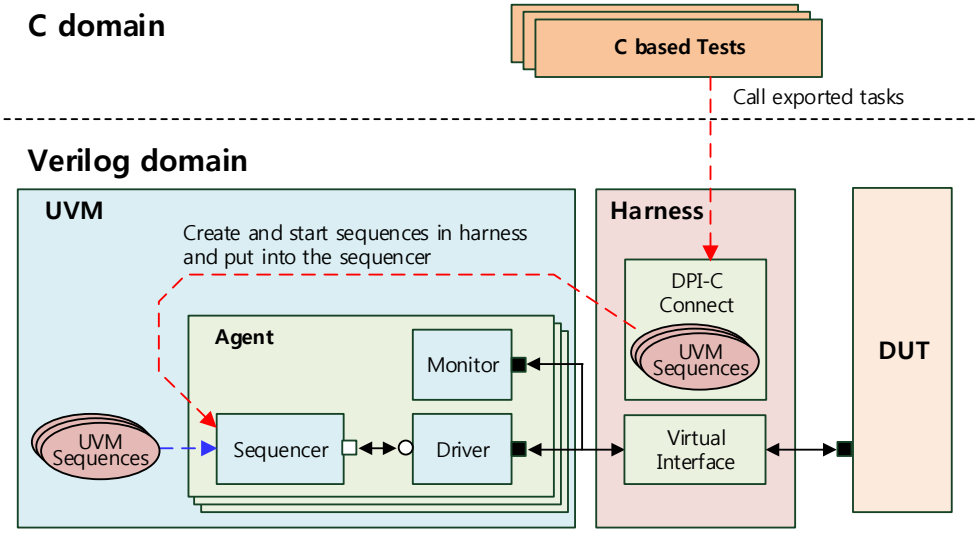

Figure 2. Testbench with DPI-C sequence and UVM environment compbined

Figure 3 shows in more detail how to combine DPI-C sequences (in C domain) and UVM environment (in Verilog domain). C-based test sequences are built in three layers in C domain so that each layer can be easily reused by other platforms. For example, the middle layer denoted *Driver* can be reused by FW test platform without a significant modification, while all the three layers are used in UVM based platform as shown in the figure. Verilog domain is split into two layers: *UVM environment* and *Harness interface*. *UVM environment* is a typical UVM dynamic component which encloses UVM sequences. These sequences call C-based test codes in C domain as shown in Figure 3. Although not shown in the figure, UVM environment has also virtual sequences which manipulate these basic sequences in various ways to create more complex scenarios, which is hard to generate in DPI-C testbench. A main function named *test_main()* in C domain is imported and called in task *run_dpic()* inside harness interface, when DPI-C test is invoked in UVM run phase. Before the run, UVM testbench and DUT are connected through an interface in UVM build phase as a typical usage of harness.

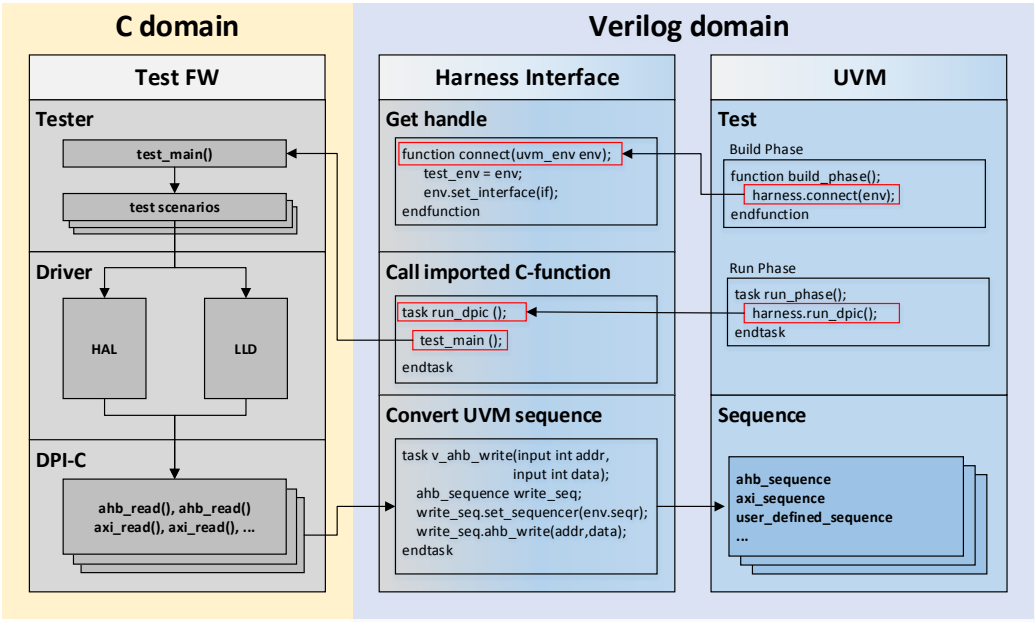

Figure 3. How to combine DPI-C sequences and UVM environment

Let us look the test flow in more detail. Two different kind files are needed to run a simulation: Verilog compiled simulation executable file and dynamically linking C shared object file. Once simulation launches, UVM test hands *uvm_env* object over to harness, during topology build phase, in which a virtual interface connects *uvm_env*, or testbench, and DUT. Such build-up can be done through *uvm_config_db* or direct assignment. Once topology configuration is completed, UVM test run_phase() calls a harness function which, in turn, calls the main function in C domain. When bus transaction task calls are encountered during the C test run, pre-defined harness version tasks are executed. Such harness version tasks feed transactions through sequencer and driver of already configured uvm_env, thereby communicating with DUT via virtual interface. When all the C tests are done, pass/fail function is called last and UVM objections are dropped to finish the entire UVM test flow.

B. Reusable Harness Interface (RHI) – Step 2

Reusable Harness Interface (RHI) is an interface format in which aforementioned harness interface can be systematically reused in a different level in hierarchy. Figure 4 illustrates the way to reuse C-based test sequences through RHI in a different level. Once developed in a lower level, the C-based test sequences can be easily reused in a higher level using RHI format, thereby reducing scenario development TAT in a dramatic fashion. Not only scenario development TAT, but debugging TAT can also be significantly reduced because error cases at a higher level can be easily reproduced at a lower level, where debugging iteration is much faster. Furthermore, considering a multi-core architected design, reusing test sequences through RHI format is much more effective in terms of TAT.

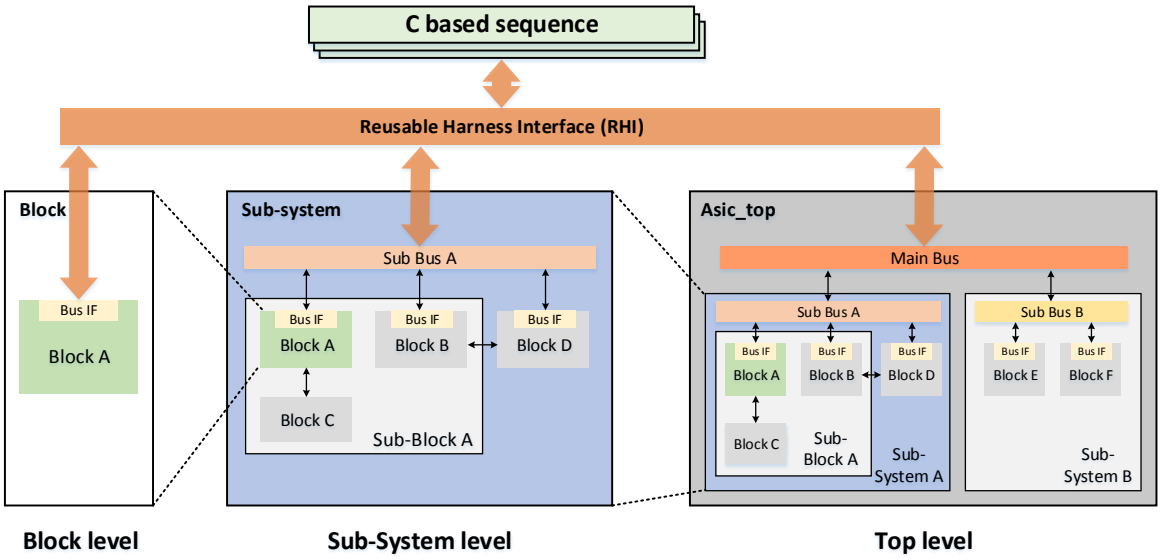

Figure 4. How to reuse C-based sequences through RHI for a different level of hierarchy

RHI consists of three major parts: *harness interface*, *harness binding*, and *path define*. *Harness interface* is conceptually the same as Harness already shown in Figure 3 in that C and Verilog domain are connected through it, but Figure 5 shows more about connecting DUT and UVM environment, which is more common use of it. Reusability of this harness is boosted up because all the DPI-C export/import functions (or tasks) are implemented inside. Such DPI-C functions convert C-based test codes into UVM sequences so that UVM environment can use C-domain test codes seamlessly.

```
/** HDL  defines */
`include hdl_defines.h

/** import components  */
import ahb_env_pkg::*;
import ahb_seq_lib_pkg::*;
```

```
/** interface declaration */a
interface ahb_harness(parameter string ENV_PATH_PARAM = "uvm_test_top.xxx.xxx")();

     /** Handle of components */
     ahb_env     dpic_ahb_env;

     /** Declaration of imported and exported functions and tasks */
     import "DPI-C" context task main(int arg0, int arg1, ... , int arg3);
     export "DPI-C" task v_ahb_swrite;
     export "DPI-C" task v_ahb_sread;
     …

     /**
     * Description imported and exported functions and tasks
     */

     task v_ahb_swrite(input int size, intput int addr, input int wdata);
        /** Convert to uvm_sequence */
        ahb_base_seq    dpic_ahb_seq;
        dpic_ahb_seq = ahb_base_seq::type_id::create("dpic_ahb_seq", dpic_ahb_env.m_ahb_mst_agt);
        dpic_ahb_seq.write(addr, size,. wdata);
     endtask : v_ahb_swrite

     task v_ahb_sread(input int size, intput int addr, output  int rdata);
        /** Convert to uvm_sequence */
        ahb_base_seq    dpic_ahb_seq;
        dpic_ahb_seq = ahb_base_seq::type_id::create("dpic_ahb_seq", dpic_ahb_env.m_ahb_mst_agt);
        dpic_ahb_seq.read(addr, size,. wdata);
     endtask : v_ahb_swrite

     /** Instantiation of virtual interface */
     ahb_interface #()
     u_ahb_interface (
        .xxx            ( `DUT_TOP.xxx   ),
        ...
        .xxx            ( `DUT_TOP.xxx   )
     );

     /** Set VIF for ENV */
     initial begin
          uvm_config_db#(ahb_interface)::set(uvm_root::get(), ENV_PATH_PARAM, "u_ahb_vif",
u_ahb_interface);
     end

     /** Get ENV component */
     initial begin
          uvm_config_db#(ahb_env)::get(uvm_root::get(), ENV_PATH_PARAM, "dpic_ahb_env", dpic_ahb_env);
     end

endinterface : ahb_harness
```

Figure 5. harness interface of RHI

Figure 6 shows *harness binding* section. Harness binding instantiates harness interface and binds it to `TB_TOP. Note that actual UVM environment instance is given through a parameter of ENV_PATH upon instantiating harness interface, whereby one can reuse the C-based test codes for a different level of hierarchy.

```
/** import components  */
import xxx_env_pkg::*;

/** HDL path defines */
`include "path_defines.sv"

/** Interface to connect with C sequence */
`include "ahb_harness.sv"

/** Binding interface */
bind `TB_TOP ahb_harness#(`ENV_PATH) u_ahb_harness ();
```

Figure 6. harness binding of RHI

Figures 7-9 show *path defines* section. Path define contains hierarchical HDL path defines per hierarchical level, e.g., IP level has a different HDL path defines from sub-system level. In addition, path define also has UVM environment path (ENV_PATH) which DPI-C translated UVM sequences are fed to.

```
/** Define design hierarchy path for block level */
`define `TB_TOP                 block_tb_top
`define `DUT_TOP                `TB_TOP.u_block

/** Define ENV path for block level */
`define `ENV_PATH               "uvm_test_top.m_block_env"
```

Figure 7. Block level path_defines in RHI

```
/** Define design hierarchy path for subsystem level */
`define `TB_TOP                 subsystem_tb_top
`define `DUT_TOP                `TB_TOP.u_subsystem.u_block

/** Define ENV path for subsystem level */
`define `ENV_PATH               "uvm_test_top.m_subsystem_env.m_block_env"
```

Figure 8. Sub-system level path defines in RHI

```
/** Define design hierarchy path for top level */
`define `TB_TOP                 top_tb_top
`define `DUT_TOP                `TB_TOP.u_top.u_subsystem.u_block

/** Define ENV path for TOP level */
`define `ENV_PATH               "uvm_test_top.m_top_env.m_subsystem_env.m_block_env"
```

Figure 9. TOP level path defines in RHI

Let us explain how to apply RHI to each different hierarchical testbench shown in Figure 4. Suppose that Block A is DUT in each testbench and there is already C test library for the block. Regardless of hierarchical level, harness interface (Figure 5) and harness binding (Figure 6) are commonly shared to test Block A. Only path define file is required to have different paths for each level. Figure 7, 8, and 9 show path define files for block, sub-system, top-level testbench respectively. If you want to build up RHI for block level testbench, all you have to do is to pick up the path define file for block level testbench (Figure 7) on top of common files of harness interface and harness binding. Similar approach can apply to other hierarchical level RHI, i.e., Figures 5, 6, and 8 for sub-system testbench RHI and Figures 5, 6, and 9 for top-level testbench RHI. Using RHI of each level, the common DPI-C test sequences are used to verify the same DUT in a different hierarchical level.

Therefore, you can run the same DPI-C test sequence of IP level at sub-system level by replacing IP level UVM environment instance path with the sub-system one without any change in other DV components. That is why we call this "Reusable Harness Interface."

IV. EXPERIMENTS

This chapter explains about experiments to measure gains from using the proposed methods. Actual tests exercised in one of the company's NAND controller projects were used under certain assumptions or conditions. Such assumptions and conditions are discussed in the next section, followed by the section showing experiment results.

A. Assumptions or Conditions

As shown in Figure 10, host protocol design of a NAND controller is chosen as DUT and UVM environment is built up with a few agents and host VIP model. DPI-C test is translated into AHB UVM sequences and they are fed to AHB agent.

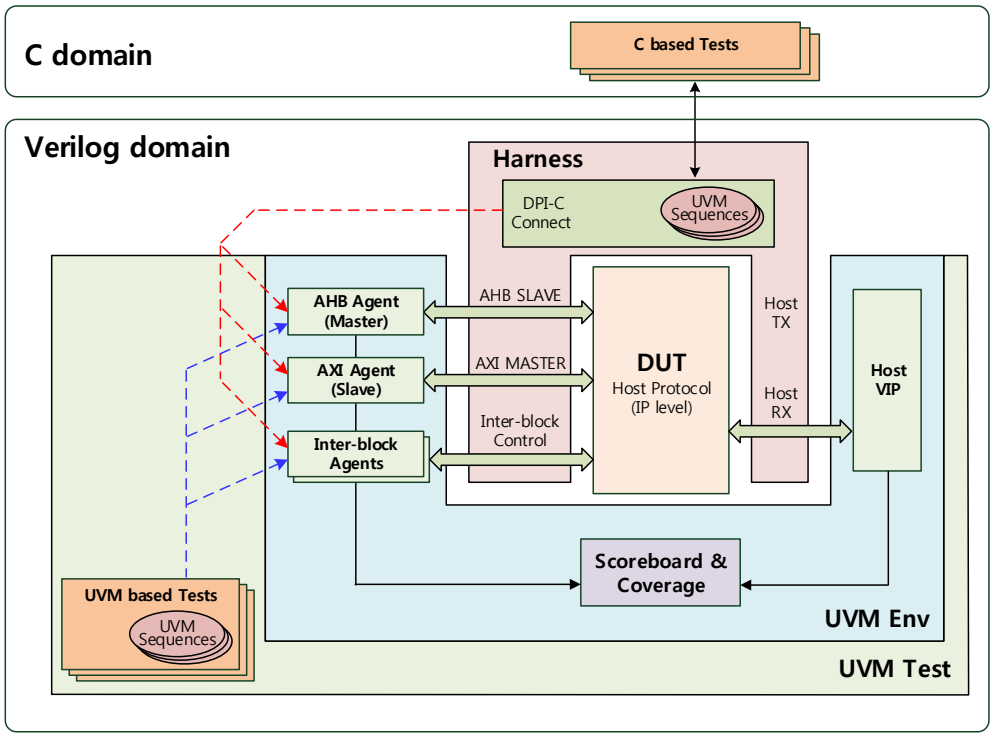

Figure 10. Block diagram for test experiments

Synopsys VCS 2016.06 is chosen as simulator. Server OS is Redhat 6.5 and only one single server is used for this test to avoid unnecessary server dependent variations affecting simulation time. Generally EDA simulators provide a certain feature not to compile unsolicited block, which have not been touched. In this experiment, due to uneven timing result depending on files we modified, we couldn't take this feature into account. The numbers shown in next section are the arithmetic average of 10 measurements. As to runtime, since the runtime difference between UVM based testbench and DPI-C/UVM combined testbench is inconsistence, we assume simulations take same run time in both cases.

Before moving on to the experiment results, let us define *Iteration Cost* as "the amount of time required to iterate once in sequence development TAT." It is assumed 0 for the time for sequence update. In a typical UVM based testbench, Iteration Cost is the sum of Verilog compile time and simulation run time, whereas Iteration Cost is the sum of Verilog/C compile time and simulation run time in our proposed testbench, or DPI-C and UVM combined one.

B. Results

Table 1 shows total sequence development TAT for each of typical UVM based testbench and DPI-C/UVM combined one. This experiment assumes 20 as the average iterations for developing a sequence with medium level difficulty by an average design verification engineer. Iteration Cost is measured about 1214sec for the typical UVM based testbench, while 805sec for DPI-C/UVM combined one. Since the average iteration is assumed 20, the total TATs are 24,280sec (= 1214sec x 20) and 16,513sec (= 805sec x 20 + 413sec) respectively, showing almost 32% TAT improvement in DPI-C/UVM combined testbench. The more complicated sequences become, the more the iteration number gets. As a result, the merged testbench can give you more benefit in TAT.

TABLE I
SEQUENCE DEVELOPMENT TAT COMPARISON

	Details	Time (sec)
UVM based	Verilog compile time x Iteration	8260
	Run time x Iteration	16020
	Total sequence develop TAT	**24280**
DPI-C + UVM based	Verilog compile time	413
	C compile time x Iteration	80
	Run time x Iteration	16020
	Total sequence develop TAT	**16513**

In general, such sequence development TAT gain becomes bigger in higher level than block level testbench. In this experiments, however, TAT gain does not increase in higher level testbench because of the run time that is becoming more dominant in total sequence development TAT. Therefore, developing test scenario in block-level testbench and applying it in higher level testbench is a time efficient way. Considering RHI, you can expect more gain in sequence development TAT which becomes more dominant at higher level testbench. In addition, debugging TAT at higher level testbench can be reduced by using lower level testbench for debugging under RHI configuration. Figure 10 compares sequence development TATs of the typical UVM based testbench and DPI-C/UVM combined one for each level.

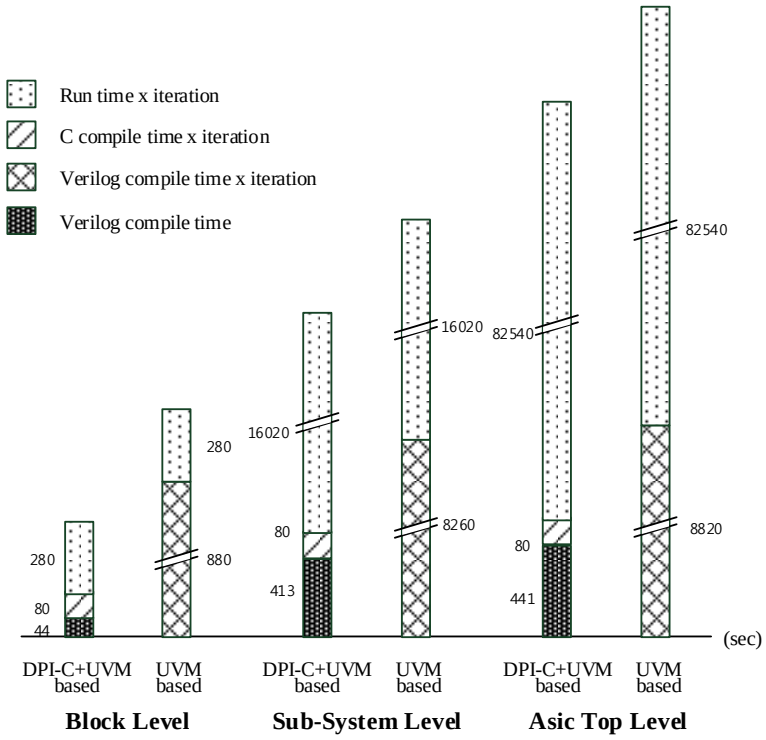

Figure 11. Sequence development TAT with different hierarchical testbench

V. CONCLUSIONS

This paper presents a practical scheme to enhance verification turn-around-time (TAT). The significant amount of time for UVM sequences development can be reduced by merging DPI-C based codes into UVM testbench via

harness interface (sequence development TAT). Such merged C-based test sequences can be used across hierarchy via Reusable Harness Interface (RHI). This makes it much easier than otherwise to reproduce higher level failing cases under lower level environment, thereby boosting up the debugging process (debugging TAT). In fact, we have achieved a dramatic reduction down to nearly one tenth of scenario development TAT and debugging TAT by using RHI in a recently completed SSD controller SoC design.

REFERENCES

[1] Harry D. Foster "*Trends in Functional Verification: A 2016 Industry Study*" DVCon, 2017
[2] Rachida EL IDRISSI, "*Guaranteed Vertical Reuse – C Execution in a UVM Environment,"* DVCon, 2013
[3] Chris Spear, "*C through UVM: Effectively using C based models with UVM based Verifiction IP,"* DVCon, 2013
[4] Rich Edelman, "*UVM SchmooVM - I want My C Test!,"* DVCon, 2014
[5] Rich Edelman, "*DPI Redux. Functionality. Speed. Optimization.*" DVCon, 2017