Efficient SCE-MI Usage to Accelerate TBA Performance

Ponnambalam Lakshmanan[1] Prashantkumar Ravindra[2] Rajarathinam Susaimanickam[3]
[1]Analog Devices, Bengaluru, India (ponnambalam.lakshmanan@analog.com),
[2,3]Aceic Design Technologies, Bengaluru, India (prashant@aceic.com, raja@aceic.com).

***Abstract*- Design complexity has been increasing with every passing day and as a result the time required to verify the same has also been exponentially increasing. Design and Verification community unanimously agrees to the fact the software simulators have not kept pace with the growing design complexity, resulting in longer simulation run time. Transaction Based Acceleration (TBA) is one of the hardware assisted acceleration techniques that has received loads of attention for its performance and ease of usage. Standard Co-Emulation Modeling Interface (SCE-MI) provides multiple use models to establish communication between the untimed software and timed hardware domains, as required by TBA architecture. This paper primarily focuses on how the inefficient SCE-MI usage dampens the TBA performance and also sheds light on how these hindrances were overcome to achieve significant acceleration. This paper discusses various efficient ways of handling SCE-MI use models for creating an efficient TBA testbench and presents practical comparison between inefficient and efficient usage of SCE-MI use models.**

Keywords— Simulators, TBA, SCE-MI, untimed software, timed hardware.

I. INTRODUCTION

Today's SoC designs are highly integrated and feature complex functionalities. The notch of complexity increases with every new design, compared to the previous generation designs, and the trend will continue for the years to come. Complex testbenches are required to thoroughly test and verify these complex designs. Verifying a design is a routine challenge to the verification engineer but what comes as a nightmare is the software simulation run time. Software simulators are pushed to the virtual limits while simulating these complex testbenches (plus the complex designs). Even though the software simulators are run on high-end computing machines, the simulation run time could vary from hours to days. As a result verification cycle engulfs a significant time share in a product cycle.

Confronted with similar challenges, we decided to explore various acceleration techniques to reduce the simulation run time. After a thorough research on the available acceleration techniques and attending few conferences/tutorials it was evident to us that TBA (explained in Section II) along with Standard Co-Emulation Modeling Interface (SCE-MI) (explained in Section III), as the communication interface, would be an ideal solution for the use case under discussion. In the course of development of the TBA testbench, an architecture was finalized (explained in section IV), which was later found to be ineffective in providing desired/expected acceleration. The root causes for the performance degradation were identified and resolved (explained in section V) and the testbench architecture was modified (explained in section VI) with respect to the usage of SCE-MI use model. The comparative result between the inefficient and efficient SCE-MI usage along with the overall performance improvement in presented in section VII. Conclusion and future work are discussed in section VIII.

II. TRANSACTION BASED ACCELERATION

Hardware assisted acceleration has always overshadowed the traditional software simulation in terms of performance but had not been adopted extensively by industry in the past; owing to a long list of limitations, such as, limited debug features, compilation time, effort to create synthesizable testbench, complex setup procedure, return on investment and so on. Things have changed, for the greater good, in the recent years with major limitations of traditional hardware assisted acceleration being addressed by the EDA vendors.

Following is the list of hardware assisted acceleration techniques:

1) *Signal Based Acceleration (SBA)*
2) *Transaction Based Acceleration (TBA)*
3) *Embedded testbench*
4) *Vector Based Acceleration (VBA)*
5) *In-circuit Emulation (ICE)*

The above list of techniques is not comprehensive by any means but lists most of the commonly/widely used acceleration techniques. SBA and TBA are often grouped under a common category called as Simulation Acceleration.

SBA is, comparatively, the easiest approach to port the test environment from software simulation to Simulation Acceleration mode. To achieve acceleration, the Design Under Test (DUT) runs on emulator and the testbench runs on software simulator. Each transition of the interface pin is considered as an event and the emulator is synchronized at every event. This obviously is not the most efficient way of utilizing the emulator. Even with this limitation SBA can provide respectable acceleration if the DUT is sufficiently large and is exercised in its entirety. The greatest advantage of SBA is that it requires no modification to the test environment.

Embedded testbench is a technique where in the entire test environment (DUT and testbench) is synthesizable and is executed on the emulator. Constructing a synthesizable testbench could be a daunting task for the DV Engineer. The unique selling proposition of this technique is the acceleration that it delivers, owing to the entirely synthesizable test environment.

TBA is a near-perfect blend of SBA and Embedded testbench techniques. The testbench follows the popular UVM structure, as in SBA, and the driving segment of the testbench is synthesizable, as in embedded testbench. It also beats the disadvantage of SBA (the performance of the emulator is not optimal in SBA because every pin toggle creates an event which increases the emulator and workstation interactions) and embedded testbench (the requirement to have the entire testbench needs to be developed using only the synthesizable constructs).

The typical architecture of TBA agent structure is shown in Figure 1. The traditional UVM agent needs to be split into two domains:

A. Non-Synthesizable domain – Runs on simulator (host machine)

Only the untimed logic must be implemented in this domain. Untimed logic includes creating constrained random stimulus/data, checker/scoreboard etc. This suits the UVM style of coding, except for the fact that the agents will neither wiggle the pins nor monitor the pin wiggles directly. This section of the code is time aware during the run. These agents are responsible for transferring/receiving the data to/from logic running on emulator over a communication interface. These non-synthesizable UVM agents are often termed as proxy agents.

B. Synthesizable domain – Runs on emulator

This domain contains the timed logic and is run on the emulator platform. The logic needs to be implemented using only the synthesizable constructs. The logic components are called as Bus Functional Model (BFM) and are responsible for converting the data received from proxy driver to pin wiggles and to monitor these pin wiggles and send the data to proxy monitor. Only BFMs should interact with the DUT. Todays advanced EDA tools allow certain non-synthesizable constructs to be run on emulator, by creating required instrumentation, but are not recommended to use due to resulting performance degradations. For better acceleration it is recommended to implement significant portion of the testbench in BFMs instead of proxy units.

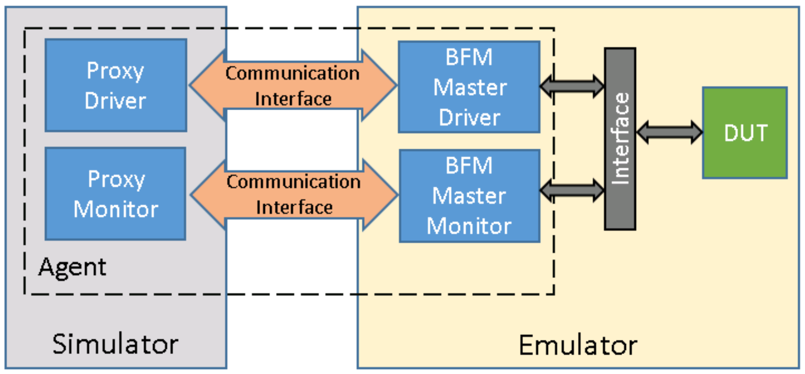

Figure 1. Typical TBA architecture

Since the testbench is split into a proxy (runs on workstation) and a BFM (runs on emulator), a communication interface is required to enable the interaction between the two units. EDA vendors provide proprietary interfaces to address this issue. Alternatively, SCE-MI (an Accellera standard) could also be used as communication interface. More about SCE-MI is explained in following section.

III. SCE-MI Use Models

SCE-MI is an Accellera standard. It is a multi-channel communication interface developed to resolve the problem of connecting/interfacing emulation platforms with the SoC modeling environments. SCE-MI use models can be used as a communication interface between the non-synthesizable agents (untimed software domain) and their respective BFMs (timed hardware domain). This is an ideal communication interface for TBA environment, especially when the testbench is based on UVM methodology.

Following are the four different types of SCE-MI use model:

A. SCE-MI Macro-based message passing interface

It comprises of a set of components both on the hardware side and the software side. The hardware side components are a set of macros which provide communication points between transactors and SCE-MI. The software side components comprise of the OOP objects and methods which perform a variety of operations. Communication interconnects are in the form of message channels operating between the host machine and emulator.

B. SCE-MI Function based interface

It is based on the concept of SystemVerilog Direct Programming Interface (DPI). It allows software and hardware side logic to be written in different languages and yet enabling them to communicate with each other (Inter-language communication). The end user is required to implement all the functions necessary to meet the required functionality.

C. SCE-MI Pipe based interface

It comprises of built-in functions, both on software and hardware side, ready to be deployed in end users code. Supports features such as batching, data shaping, variable length messaging, end of message indicator and flushing. It is ideal for streaming applications. It has the concept of input pipe (to transfer data from software side to hardware side) and output pipe (to transfer data from hardware side to software side). These are basically unidirectional.

D. SCE-MI Direct memory interface (DMI)

It provides software side interface to perform backdoor read/write operations on the hardware side memories. It basically provides two types of interfaces: Block interface and Word interface. These APIs provides a non-intrusive C-side interface to directly access a BFM's memory or a register at a single instance in simulation time.

IV. Architecture Before Optimization

The goal was to develop an agent that supports both "Software Simulation" mode and "Simulation Acceleration" mode. Since the agent has to be developed on UVM guidelines, it was ideal to opt for TBA technique. As required by the TBA technique the agent was split into two segments, one being the non-synthesizable domain (UVM environment) and the other being the synthesizable domain (BFMs). SCE-MI pipes were shortlisted as the communication interface between the two domains. The testbench had two agents, the master and the slave, and hence to test, these agents they were integrated back-to-back as shown in Figure 2. Figure 2 is a simplified block diagram to show the high-level architecture of the testbench under discussion.

Following is the brief list of issues related to SCE-MI pipes that significantly degraded the accelerator's performance.

A. Multiple SCE-MI pipes

HDL side implementation of the SCE-MI pipe is not purely synthesizable and hence there are some behavioral evals associated with it, depending on the usage. The evaluations performed by the emulator on the non-synthesizable code (Vendors support some of the non-synthesizable constructs to be implemented on emulators) is termed as behavioral evals. In the initial architecture, the required number of instances of master BFM to be created was based on a parameter (Ex: If the parameter value was 10 then ten instances of master BFM were instantiated) and we had multiple such instances to be integrated as the project requirement. Each instance had its own input

SCE-MI pipe to receive data from the proxy driver. The total number of behavioral evals was the summation of behavioral evals associated with the pipe of all the instances. Increase in the number of behavioral evals resulted in the degradation of the acceleration performance.

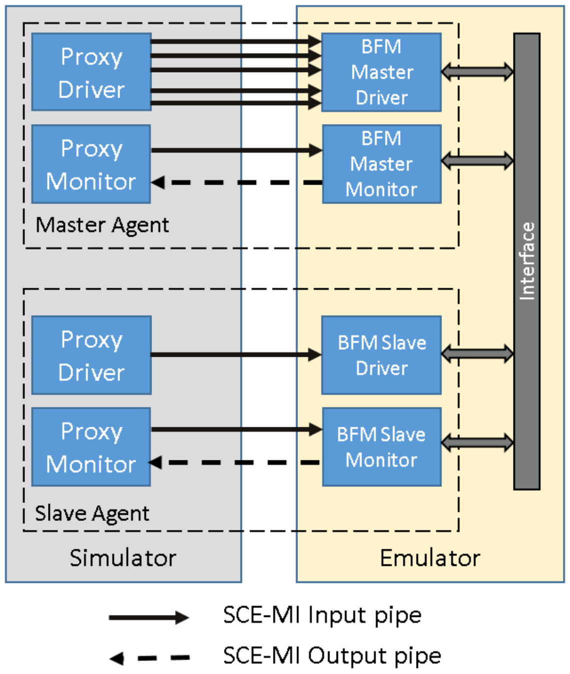

Figure 2. Simplified testbench architecture before optimization

B. Clearing the contents of the pipe - Asynchronous access

It was required to clear the contents of the SCE-MI pipes on reset, to ensure that the unwanted data of previous reset cycle does not pollute the data of the current reset cycle. To achieve this functionality, the input SCE-MI pipe was used inside a loop. This loop extracts the elements from the pipe until the End of Message (EOM) bit is “1”. This ensured that the remaining elements in buffer are cleared on reset. The pipe, used in the loop, was configured to operate in asynchronous mode. The SCE-MI pipe in asynchronous mode of operation exhibit behavioral evals (as it is not purely synthesizable) and the loop logic results in increased step count. These factors degraded the acceleration performance.

C. Quantity of data accessed

The data was split into bytes in the proxy for ease of debug. The BFM was expected to collect multiple such bytes and form the final data. It was attempted to fill the pipe by configuring BYTES_PER_ELEMENT as one byte and PAYLOAD_MAX_ELEMENTS as multiple bytes. Accessing one byte per element resulted in poor accelerator performance.

D. Synchronizing simulator and emulator

The emulator operates at much higher frequency as compared to the simulator. To avoid loss/overriding of data, the simulator and the emulator were required to synchronize (halt the emulator) at frequent intervals (whenever the buffer was full/empty in BFM). The overall run time in TBA mode increases as the number of synchronization between the timed and untimed domains increases, resulting in reduced acceleration.

V. Efficient SCE-MI Usage

As discussed in section III there are multiple use models at ones disposal to be used at ones requirement. In the initial architecture SCE-MI pipe was the only use model been used to establish communication link between the timed and untimed domains. SCE-MI use models support co-existence, different use models could be used in single testbench. This was not explored in the initial architecture. This section discusses the workarounds and changes been made in the testbench architecture to overcome the challenges discussed in precious section.

A. *Power of back-door access*

Direct Memory Interface (DMI) is a powerful SCE-MI use model when it comes to back door access of registers and memories. Instead of transferring the data frequently to the proxy, a memory could be created in the BFM to accumulate the data. Once a reasonable amount of data was buffered, SCE-MI DMI API could be used to read the whole memory from the proxy domain effectively resulting in the performance improvement. Figure 3 shows an example modelling of the C-function, which uses SCE-MI API to read the memory from the BFM. The sole negative trait of the DMI is its inability to synchronize simulator and emulator, resulting in a potential loss of data.

```
//C-function which reads from the HDL memory
//This function also returns the contents of the memory back to the proxy side
void read_mem(svBitVecVal return_mem[8192]){
   static void* vmem
   int raddr;
   static unsigned int width, depth;
   vmem = scemi_mem_c_handle("hierarchical_path.memory");
   scemi_mem_get_size(vmem, (unsigned int *) &width, (unsigned int *) &depth);
   scemi_mem_get_block(vmem, raddr, depth, return_mem);
}
```

Figure 3: C-function with SCE-MI DMI implementation

B. *Optimized SCE-MI pipe usage*

Although SCE-MI pipes usage contributed towards the overall behavioral evals, its ability to synchronize the simulator and the emulator made it hard to avoid. The solution would be to tactically optimize its usage, where it exhibits less behavioral evals.

1) *Merging SCE-MI pipes: If m*ultiple SCE-MI input pipes are transferring same type of data from proxy to BFM then the functionality of these pipes could be merged into a single pipe which would be fed with concatenated data required for all instances in the BFM. The wrapper module would have the HDL side handle of the SCE-MI pipe which would fetch the data from proxy and distribute it between the instances of the required module. This would streamline the operation and reduce the behavioral evals.

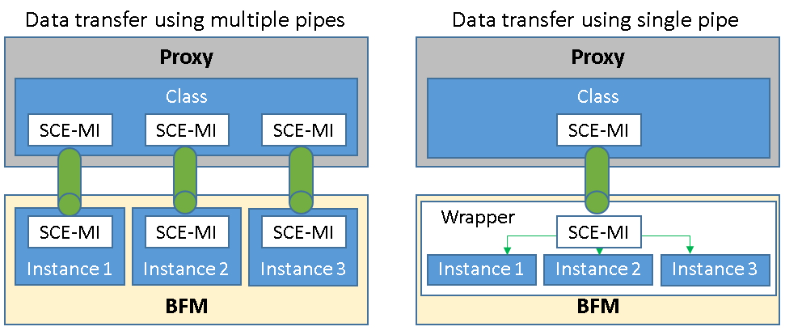

Figure 4: TB Architecture with merged SCE-MI pipe

2) *Synchronous access:* Asynchronous access to the pipes leads to behavioral evals. It is recommended to use SCE-MI pipes in synchronous mode (clocked pipe) to avoid behavioral evals by configuring the parameter IS_CLOCKED_INTF of the SCE-MI pipe to one, as shown in the Figure 5. This ensures that the data access happens only on the posedge or negedge of clock depending on the sync_control setting in the respective calls.

```
//Input pipe as the direction of transfer is from proxy to bfm
scemi input pipe #(.BYTES PER ELEMENT(1),    //Total bytes to be transferred at a time(1 Byte)
                   .PAYLOAD MAX ELEMENTS(1), //Elements transfered in the buffer(1 Element)
                   .VISIBILITY MODE(2),
                   .IS CLOCKED INTF(1)       //Clocked
                  ) inbox(device clk);
```

Figure 5: Synchronous SCE-MI pipe

3) *Optimized data transfer:* Instead of transferring “N” elements, with size of each element being a byte, the data is recommended to be combined and sent as one element of size “N” bytes. This implementation would effectively reduce SCE-MI access by “N” times thereby boosting the performance. A sample code snippet of the SCE-MI pipe declaration is shown in Figure 6.

```
//Input pipe as the direction of transfer is from proxy to bfm
scemi input pipe #(.BYTES PER ELEMENT(5),    //Total bytes to be transferred at a time(5 Bytes)
                   .PAYLOAD MAX ELEMENTS(1), //Elements transfered in the buffer(1 Element)
                   .VISIBILITY MODE(2),
                   .IS CLOCKED INTF(1)       //Clocked
                  ) inbox(device clk);
```

Figure 6: Size of each element in a SCE-MI pipe

4) *Minimal Synchronization:* Data transfer between proxy and BFM causes simulator and emulator to synchronize under the hood. Acceleration could be increased by avoiding frequent data transfers. As an example, instead of sending one element of size five bytes at regular intervals, twenty elements could be packed together to reduce the simulator – emulator synchronizations.

```
//Input pipe as the direction of transfer is from proxy to bfm
scemi input pipe #(.BYTES PER ELEMENT(5),     //Total bytes to be transferred at a time(5 Bytes)
                   .PAYLOAD MAX ELEMENTS(20),//Elements transfered in the buffer(20 Elements)
                   .VISIBILITY MODE(2),
                   .IS CLOCKED INTF(1)        //Clocked
                  ) inbox(device clk);
```

Figure 7: Maximum number of elements in a SCE-MI pipe

5) *Clearing the pipe:* At times it might be necessary to clear any residual data present in the pipe upon reset or interrupt. Instead of using for-loop to clear the pipe, as shown in Figure 8, it is recommended to use synchronous procedural block to achieve the same functionality. A clock, faster than the operational clock, would be required to clear the pipe synchronously. This avoided all the behavioral evals and step counts due to the for-loop. The end of message (eom) mechanism with this procedure could be coupled to signal the end of the residual data. The logic in Figure 8 uses for-loop to extract data from the pipe whereas the logic in Figure 9 uses a faster clock (frequency of clk_fst > clk) to do the same. Use of faster clock helped avoiding the for-loop thereby reducing the step-count and behavioral evals.

```
//Inefficient way of clearing the pipe using a for-loop
always@(posedge clk, posedge rst) begin
   if(rst) begin
      for(int i=0; i<20; i++)
         inbox.receive(1,ve,data,eom);
   end
end
```

Figure 8: Clearing a SCE-MI pipe, using for-loop

```
//Faster clock used to synchronously clear the pipe
always@(posedge clk_fst, posedge rst) begin
   if(rst)
      rst_buff = 1;
   else begin
      if(rst_buff && !eom) inbox.receive(1,ve,data,eom);
      else rst_buff = 0;
   end
end
```

Figure 9: Synchronously clearing the SCE-MI pipe

C. *Savaging Direct Programming Interface*

Function Based Interface is bundled with the ability to establish a communication interface between the hardware side and the software side using DPIs. It also gives the liberty to write one’s own functions, as per the required functionality, in one language and invoke it from another. Figure 10 shows an example c-function, which uses DPI to invoke a method defined in BFM. Scope to this BFM is set using the function svSetScope. This method is used to configure a register in the BFM.

```
//C-function has the implementation of proxy's write function
//It also invokes the BFM's write function throuh it
const char* tbpath = "path_to_the_bfm_where_reg_write_function_is_defined"
static svScope       tbscp = NULL;
extern void reg_bfm_wr( svBitVecVal* reg_data);
void write_bfm_reg_c(svBitVecVal* reg_data){
   tbscp = svGetScopeFromeName(tbpath);
   svSetScope(tbscp);
   reg_bfm_wr(reg_data);
}
```

Figure 10: C-function invoking BFM's method

Defined C-function can be imported and invoked from the proxy using DPI as shown in the Figure 11. This in turn executes the method defined in the BFM. As per this example, data is passed through the C-function to be written into the BFM's register.

```
//Proxy code, here is where the write data is passed to the C-Function which is to
//be written on to the BFM
import "DPI-C" context write_bfm_write_c = function void write_bfm_reg(bit [31:0] data);
class proxy_driver;
   bit [31:0] data_wr;
   task reconfig;
      write_bfm_reg(data_wr);
   endtask
endclass
```

Figure 11: Proxy using DPI to invoke C-function

The method implemented in the BFM is exported using the inbuilt DPI, as shown in the Figure 12, to be accessed by the proxy through the C-function. Hence the register write invoked by the proxy gets executed in the BFM through the C-function which are facilitated by the DPI calls.

```
//BFM code, here is where the implementation of the write function resides
//It is invoked by the C-interface
module BFM;
   export "DPI-C" function reg_bfm_wr;
   bit [31:0] ctl_reg;

   function void reg_bfm_wr(input bit [31:0] wr_data);
      ctl_reg = wr_data;
   endfunction
endmodule
```

Figure 12: BFM exports its method through DPI

D. *Trimming TBA development time using SCE-MI*

As synthesizable BFMs are the integral part of the TBA architecture, it mandates its functionality to be implemented in an optimized synthesizable form. Implementation of these functionality by a DV Engineer could impact the performance because of the following factors:

- Increase in effort and time to implement complex functionality in a synthesizable format.
- Increase in area occupied by the BFM on the emulator.

Proper utilization of SCE-MI DPI functions to implement any complex functionality will help to reduce the above factors. Although it requires a proper understanding of how to map the C data type with the SystemVerilog data type, the benefit it offers outweighs the effort. Any complex functionality, for example: error correction algorithm, which is utilized only during an occasional error injection test case is implemented as a method in the C membrane and is imported/invoked from the BFM. Implementing basic BFM functionality as a C-function will result in performance degradation, hence it would be wise to model complex and rarely utilized functionality as C-function. Packed array is one of the widely used data type to classify data in SystemVerilog, but mapping it to its C counterpart is quite tricky. Below mentioned are couple of useful attributes related to the mapping of a packed array with *svBitVecVal** or *svLogicVecVal**, which will save a definite amount of bring up effort.

1) *Packed array – pass by reference*: As per the SystemVerilog standard, *inout* and *output* arguments, with the exception of unpacked arrays, are always passed by reference. Specifically, packed arrays are passed,

accordingly, as *svBitVecVal** or *svLogicVecVal**. Hence meddling with these arguments in C will naturally have its effect on its SystemVerilog counterpart (BFM).

2) *Dissecting a packed array*: A packed array is passed as a *svBitVecVal** or *svLogicVecVal** type to be used in a C – function. But mapping this is not straight forward, a svBitVecVal* or svLogicVecVal* acts like a dynamic array with each element of size 32-bit. It implies, if a packed array of size more than 32-bit is mapped to a *svBitVecVal** or *svLogicVecVal** type then it is broken in to chunks of 32-bit data and each 32-bit is mapped to an element of *svBitVecVal** or *svLogicVecVal** starting from packed array's LSB. An example comparison of a bit-select operation performed in a BFM and a BFM invoking a C-function is shown in the Figure 13 and Figure 14 respectively. Figure 13 shows the BFM implementation of the bit-select operation of a packed array. In the Figure 14, the same bit-select operation is implemented as a C-function *bit_sel_c*. It is imported in the BFM and mapped to its *bit_sel_im* function. Hence by invoking *bit_sel_im*, *bit_sel_c* in C will be executed. Figure 14 shows how a packed array in a BFM is mapped to *svBitVecVal** type in the C-function.

```
//Preforming a bit select in SV
bit [63:0] a_sv;
always@(posedge clk) begin
   //bit 63 of a_sv is assigned to selc_bit
   selc_bit = a_sv[63];
end
```

Figure 13: BFM doing bit select operation

```
//Preforming a bit select using C -DPI
//BFM part of the code
bit [63:0] a_sv;

import "DPI-C" context bit_sel_c =
function bit bit_sel_im(output bit [63:0] a_sv);

always@(posedge clk) begin
   //bit 63 of a_sv is assigned to selc_bit
   selc_bit = bit_sel_im(a_sv);
end
```

```
//C - Part of the code
svBit bit_sel_c(svBitVecVal* a_c) {
   //Mapping between bit [63:0] and svBitVecVal* is
   //a_c[0] = a_sv[31:0] and a_c[1] = a_sv[63:32]
   //bit 63 of a_sv corresponds to a_c[1]'s bit 31
   unsigned int selc = mask_bit_sel(a_c[1],31);
   return selc;
}
//Customized function to do the bit-select
svBit mask_bit_sel(unsigned int  input_dat, unsigned int position){
   unsigned int mask = 1 << position;
   unsigned int masked_n = input_dat & mask;
   unsigned int thebit =  masked_n >> position;
   return thebit;
}
```

Figure 14: BFM doing bit select operation using a C – DPI call

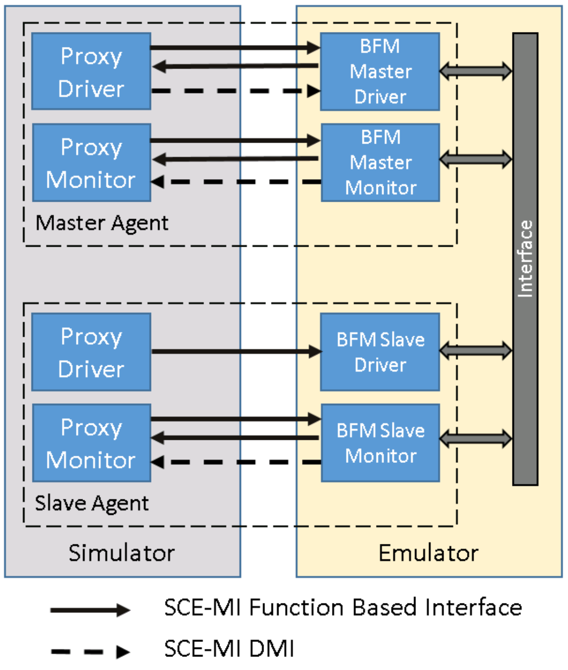

Figure 15. Simplified testbench architecture after optimization

VI. Architecture After Optimization

The simplified architecture of the testbench after performance optimization is as shown in Figure 15. The SCE-MI Function Based Interface was used to configure the registers of the BFMs of both master and slave agent. These were also used to provide certain notifications (about relevant protocol events) to the proxy agents from the BFMs. APIs of SCE-MI DMI were used to perform backdoor read and write operations. EDA vendor proprietary options were used to establish the synchronization between the software and hardware domains.

VII. Results

This section of the paper presents the statistical comparison between the TBA performance before and after the optimization. Finally it also presents the direct run-time comparison between the software simulation mode and TBA mode.

Both the versions of the agent, before and after the optimization, were subjected to the same test case (same configuration and same seed) to gauge the true difference in the TBA performance. It was obvious that the test environment with optimized agent would perform better compared to predecessor, but the real intent was to gauge the return on investment (effort put into optimization v/s performance improvement). The comparative results are encouraging as shown in Table 1. The optimized agent performance is better by an order of fifteen. As shown in Table 1 the optimized agent had very minimal behavioral evals and reduced hardware-software synchronization, these two factors significantly influence the TBA performance.

TABLE I
Comparative results of the implementations: before and after optimization

Implementation	**TBA properties**			
	Gate count	***Bevals***	***HW-SW Sync***	***TBA Time***
Before optimization	~2 M	18,299,173	4,728,998	~60 min
After optimization	~1.5 M	218	9,871	~4 min

The block level run with the optimized agent, which took hours in software simulation, was reduced to mere minutes in TBA mode. As shown in Figure 16 acceleration of 45X was achieved with the TBA against the software simulation. The acceleration is expected to improve once the DUT is integrated in the test environment, as it is entirely made of synthesizable constructs.

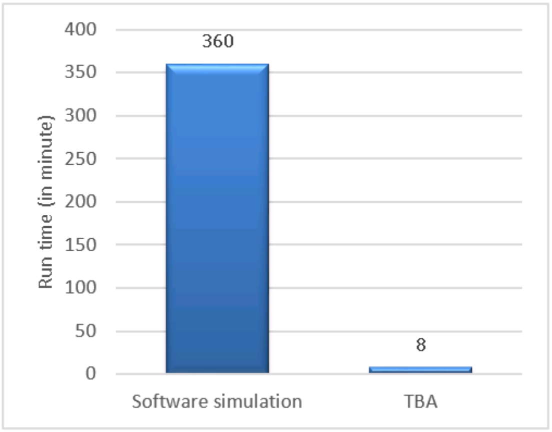

Figure 16. Software simulation vs TBA run-time comparison

VIII. Conclusion and Future Work

Hardware assisted acceleration techniques are the most advanced and highly utilized techniques in the DV Engineers arsenal. For obvious reasons, setting up the TBA testbench is not as straight forward as it is with the software simulation testbench. In this paper we illustrated the adverse impact on the TBA performance by careless use of SCE-MI use model (in particular, the usage of SCE-MI pipe) and also proposed workarounds for the issues encountered in the course of development of the efficient TBA architecture. Testbench modifications before and after optimization were discussed in-detail to assist the engineers to apply these optimization techniques up-front in their testbench development cycle.

As discussed throughout the paper, excessive hardware software synchronizations and behavioral evals has continued to remain as the major threat to the TBA performance. All the efforts been putforth in the development process of the TBA testbench would not result in "real value add" if appropriate SCE-MI use-models are not deployed. In this work, SCE-MI DMI was used for backdoor read/write operation, SCE-MI Function Based Interface was used to write to registers and to send notifications to proxy from BFM. Hardware Software syncs were generated using vendor proprietary implementation and the number of syncs were brought down to the minimum. Although we avoided using SCEMI pipes completely to gain performance in our application, it is generally recommended to use SCE-MI pipes for streaming application requiring advanced controls such as data shaping, variable length messaging and others.

As part of the future work, we have planned to integrate the optimized agent with the DUT, to verify the DUT's functionality on the emulator. Furthermore, the agent will be eventually ported to the SoC environment. Also the latest SCE-MI version (v2.3) provides additional support to UVM register access which are yet to be explored.

REFERENCES

[1] Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual, Version 2.2 Accellera, Jan 2014.

[2] Standard Co-Emulation Modeling Interface (SCE-MI) Reference Manual, Version 2.3 Accellera, Jun 2015.

[3] IEEE STANDARD 1800-2012 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language, IEEE, Feb 2013.

[4] Hans van der Schoot and Ahmed Yehia, "UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up", DVCon Europe – 2015