2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Enhancing Transaction Based Acceleration Performance Using Efficient SCE-MI Modelling

Ponnambalam Lakshmanan, Analog Devices, Bangalore, India
(ponnambalam.lakshmanan@analog.com)

Prashantkumar Ravindra, Aceic Design Technologies, Bangalore, India (prashant@aceic.com)

Rajarathinam Susaimanickam, Aceic Design Technologies, Bangalore, India (raja@aceic.com)

Anilkumar TS, Cadence Design Systems, Bangalore, India (anilts@cadence.com)

***Abstract*— An ever increasing simulation run time has forced us to move towards hardware assisted testbench acceleration techniques. One such noteworthy technique is the Transaction Based Acceleration (TBA). It has been proven to deliver good performance improvement by modeling the simulator – emulator interactions as transactions. SCE-MI does the strenuous job of interfacing the untimed software with the timed hardware. Inefficient usage of SCE-MI directly affects the acceleration as it severely degrades the TBA performance. In this paper the efficient SCE-MI modeling techniques in conjunction with various use-models by which the acceleration performance can be enhanced are discussed.**

Keywords*— *Acceleration, TBA, SCE-MI, emulator.

I. INTRODUCTION

Hardware assisted testbench acceleration setup is compelling in recent times owing to the ever growing size of the design and the time taken to thoroughly verify it. Multiple acceleration techniques are at one's disposal today, out of which Transaction Based Acceleration (TBA) has been proven to provide significant increase performance improvement over its counter-part by addressing issues such as minimizing the simulator – emulator interactions (drawback in Signal Based Acceleration- SBA) and constrained-random stimulus (drawback in Embedded Testbench). Its ability to provide a near-perfect blend of the features of aforementioned techniques has made TBA the most sought acceleration technique. Reduction in HW-SW synchronizations can be effectively achieved by following the below steps:

A. Splitting the testbench into two domains

- Synthesizable domain: Modeling a part of the testbench using synthesizable constructs. This section of the testbench can be termed as Bus Functional Model (BFM). It is responsible for driving the data that it has received from the non-synthesizable (proxy) domain onto the interface. This is the pin wiggler. This along with Design Under Test (DUT) runs on the emulator platform.
- Non-synthesizable domain: This section of the domain is commonly termed as proxy. It is responsible for creating the constrained random stimulus/data and pass it to the BFM, rather than driving the interface by itself. It runs on the simulator like a conventional testbench does.

B. Modeling the software - hardware interactions

The communication between software (simulator) and hardware (emulator) is modeled as transaction based, not cycle/event based, to reduce the communication overhead between the two. Standard Co-Emulation Modeling Interface (SCE-MI) provides a bunch of communication interfaces to realize the data transfer between the software and the hardware. More about these interfaces are described in the next section. A typical TBA architecture utilizing the SCE-MI interface (pipes) is shown in Figure 1.

Although Accellera has standardized SCE-MI to realize the communication between the software and the hardware domain, the acceleration efficiency largely depends on its usage. Improper usage of SCE-MI interfaces significantly degrades the acceleration performance, making it virtually ineffective.

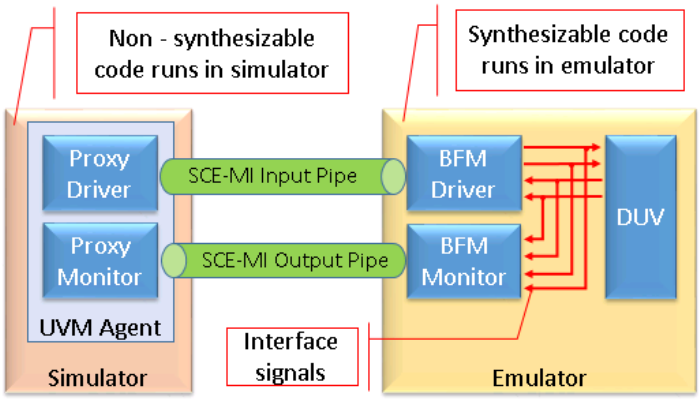

Figure 1: Typical TBA Architecture

In this paper some of the effective ways of modeling the SCE-MI interactions and its impact on the acceleration performance has been explored. SCE-MI v2.2 has been used for the described work. A use-case of our implementation is described in this paper, where the acceleration performance was enhanced by efficient usage of the SCE-MI interface. Results are compared in the later section to showcase these improvements.

II. SCE-MI AND ITS FLAVOURS

SCE-MI is essential for seamless communication between the simulator and the hardware accelerator. This eliminates the uncertainty of its operation across emulators, from various vendors, but it might also be a subject of malicious usage. For DV engineers, who are comfortable with System Verilog and UVM based testbench approaches, transferring transactions back and forth across components is a common way of data transfer. SCE-MI makes DV engineers feel comfortable, as it allows transactions to be used to communicate between BFM and proxy part of the testbench. Since the idea in TBA is to exchange transactions between the BFM and the proxy, SCE-MI is a natural choice, as it is designed to realize just this. Transaction in this context is a data structure, which the synthesizable BFM can understand. The transaction can be a group of bytes packed in a definite order packed by the proxy and sent to the BFM to be processed and converted to pin wiggles on DUT's interface. SCE-MI interfaces can be broadly categorized in to four flavors:

A. Macro-based Message passing Interface

Macro based message passing interface acts as a message channel that runs between the proxy and the BFM. Each message channel has two ends. The end on the proxy side is called the message port proxy, which gives API access to the channel. The other end on the BFM side is called message port, which is instantiated within the BFM. It has the ability to freeze controlled time while performing message composition and decomposition from the BFM.

B. Pipe-based Transaction

Pipe-based transaction uses transaction pipes to do variable length messaging and streaming data across the proxy and the BFM. A transaction pipe is a construct which is accessed via function calls to do this data streaming. Transaction pipes are unidirectional with the transaction always flowing in one direction. The pipes are termed either as an input or an output pipe based on the direction of the transaction with respect to the BFM. Pipe-based transactions has known advantages such as data shaping, end-of-message marking mechanism and support for multiple pipe transactions in zero time.

C. Function-based Interface

It savages the DPI functionality of System Verilog allowing inter-language communication without any fuss. Hence one can define her/his own API by defining functions in one language and calling them from the other. In this interface a function call is termed as a transaction.

D. Direct Memory Interface (DMI):

Provides two predefined API's for backdoor memory and register access. 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. The SCE-

MI also leverages the existing standardized register API that is part of the Universal Verification Methodology (UVM) for register access.

III. SCE-MI PIPES – MALICIOUS USAGE

The RTL, for which verification environment was developed, had master and slave components associated with it. The goal here was to create a master and a slave agent to verify the RTL. Furthermore, these agents were meant to support acceleration. As per the TBA methodology, the agents and their respective components, driver and monitor, were partitioned into non-synthesizable proxies and synthesizable BFMs. The communications between these proxy and BFM components were modelled using SCE-MI pipes due to its known advantages, as described in the previous section. The raw architecture before optimization is as shown in the figure 2.

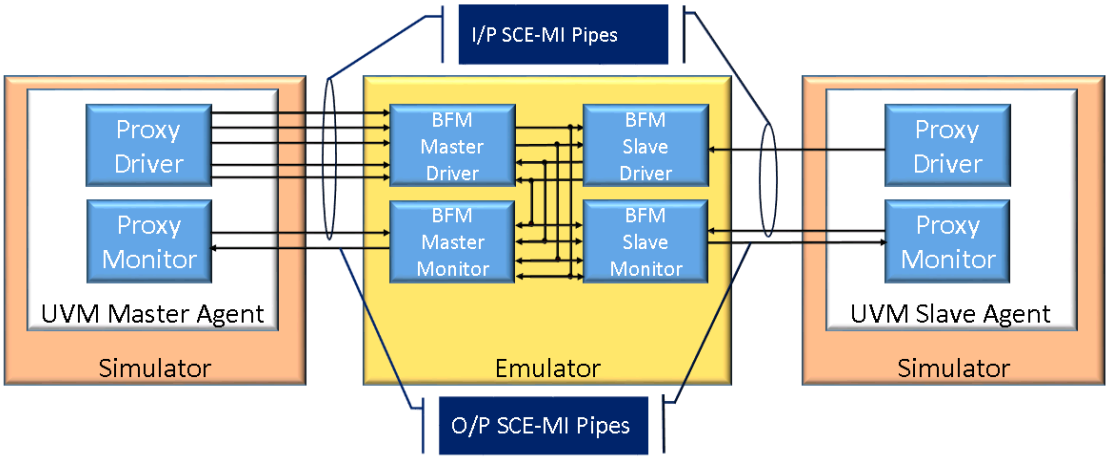

Figure 2: TB Architecture before optimization

The intended functionality was achieved by connecting these agents back to back and porting it to the emulator. At that point, primary focus was to achieve the functionality and eventually the same was achieved. The focus, then shifted towards the main reason for opting TBA approach - "The Acceleration". This is when the malicious usage of the SCE-MI pipes was noticed. It affected the performance to a level where the TBA performance was meekly better than the simulation only mode. Well, we had to put-on our detective hat and hunt for faulty usage. Before long we realized that there were multiple reasons behind the degradation of speed, of which, the main culprit was the way we used the SCE-MI pipes. After maneuvering through the issues, we concluded on the following classification of the performance degraders.

A. Multiple SCE-MI pipes

SCE-MI pipe, when mapped to the hardware, exhibits certain amount of behavioral operations. These behavioral operations contribute towards the behavioral evals. Acceleration is inversely proportional to the amount of behavioral evals. As the requirement was to transfer random data in parallel to multiple instances of the same module, each instance had its own SCE-MI pipe. As the number of instances of the said module increased, the behavioral evals associated with SCE-MI instance also increased, leading to degraded TBA performance.

B. Asynchronous access

Usage of the unclocked SCE-MI pipes (asynchronous mode) added up to the behavioral evals count. Additionally SCE-MI pipe was accessed in a for-loop in the BFM, to fetch more than one element from the proxy at a given instant of time. This resulted in behavioral evals degrading the performance.

C. Amount of data accessed

The data was split into bytes and this resulted in accessing more than one element from the pipe to frame a particular message in the BFM. It was attempted to fill the pipe by configuring BYTES_PER_ELEMENT as one byte and PAYLOAD_MAX_ELEMENTS as more than one byte. This led to accessing the pipe multiple times in the same cycle amounting to performance degradation.

D. *Synchronising simulator and emulator*

To avoid loss/overflow of data, the proxy and the BFM must be synchronized before the data transfer happens. The operation of the emulator gets stalled during the process of synchronizing proxy and BFM for seamless data transfers. Whenever the emulator stalls, it leads to surge in the TBA run time. Thus when the synchronization is frequent, the stall is frequent and hence the performance degrades.

E. *Clearing the pipe*

At reset or during an interrupt there might be a residue of data left in the SCE-MI buffer which might no longer be valid. The requirement was to clear the residual contents of buffer before BFM requires a new set of random data from proxy. This required the contents of the pipe to be cleared asynchronously using for-loops, eventually leading to increase in behavioral evals.

IV. TAMING SCE-MI

After extensive research, few approaches were identified which soothed our minds towards acceleration. Important one is co-existence, a team work between various SCE-MI use-models aided in achieving the desired goals. The optimization efforts in improving the acceleration performance can be characterized as below.

A. *Power of back-door access*

Direct Memory Interface (DMI) is a beast when it comes to back door access of registers and memories. The sole negative trait of the DMI is its inability to synchronize simulator and emulator, resulting in a potential loss of data. In order to effectively use DMI, pipe based interface was utilized. This pipe aided in achieving the required synchronization. This approach resulted in an implementation which proved to be effective when it comes to data transfer between proxy and BFM. A memory was created in the BFM and data was accumulated instead of transferring it frequently to the proxy. Once a reasonable amount of data was buffered, SCE-MI pipe was used to synchronize and SCE-MI DMI API 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-MIAPI to read the memory from the BFM.

```
//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 long long *)&depth);
    scemi mem get block(vmem, rAddr, depth, return mem);
}
```

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

The Proxy would import the user defined function and make use of it to read the intended BFM's memory as shown in the figure 4. Synchronization has been done using SCE-MI pipe before and after doing the back-door read.

```
//proxy-side class, which imports the C-function
//It uses the C-function to read the memory through back door
//Starts to read after sync through SCE-MI pipe
import "DPI-C" task read mem(output bit[7:0] return mem[8192]);
class proxy monitor;
   bit [7:0] return mem [8192];
   task run test();
   scemi pipe receive.receive bits(1,valid,start receive,eom);
   read mem(return mem);
   scemi pipe receive.receive bits(1,valid,mem received,eom);
   endtask
endclass
```

Figure 4: Proxy-side implementation of SCE-MI DMI

Figure 5 shows the implementation of the BFM memory and the way the data is accumulated in the same. Synchronization is established with the proxy through the SCE-MI pipe, indicating the proxy that the memory is ready to be read.

```
//BFM-side module, this implements the memory to store the data
//It signals the proxy on when to read the data from the memory
module monitor bfm;
   always @(posedge clk) begin
      mem[wr ptr] = data;
      if(wr ptr == FULL) begin
         scemi pipe send.send(1,ready to send,1);
         scemi pipe send.flush(1);
         scemi pipe send.send(1,mem read,1);
         scemi pipe send.flush(1);
      end
      else wr ptr = wr ptr++;
   end
endmodule
```

Figure 5: BFM-side implementation of the memory and sync procedure

Figure 6 pictorially shows the way in which the SCE-MI pipe is used to establish sync between HW-SW to ensure the BFM's memory is read by the proxy before the memory is overwritten.

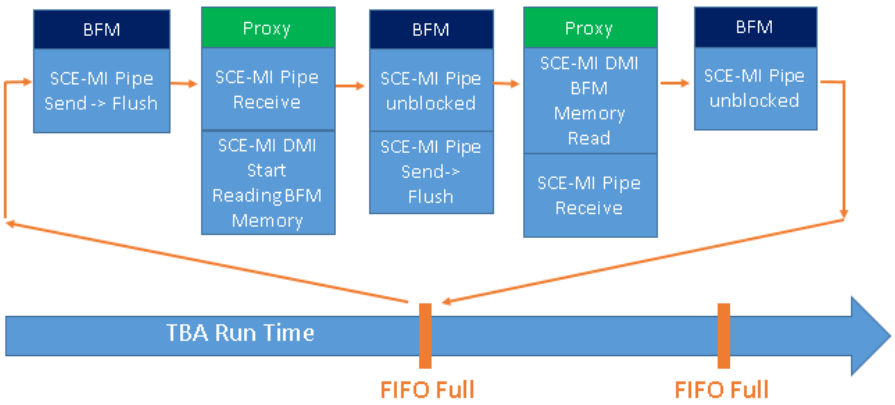

Figure 6: HW-SW sync and data transfer

B. Optimized 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 idea was to tactically optimize its usage, where it exhibits less behavioral evals. Following is the list of issues addressed.

- Merging SCE-MI pipes: In the original implementation individual SCE-MI pipes were transferring data from different instances of proxy to their respective BFM instances. As it increased the behavioral evals, it was decided to merge these pipes to a single instance so as to enable data gathering and splitting at their respective ends as shown in the figure 7. This streamlined the operation and reduced the behavioral evals.

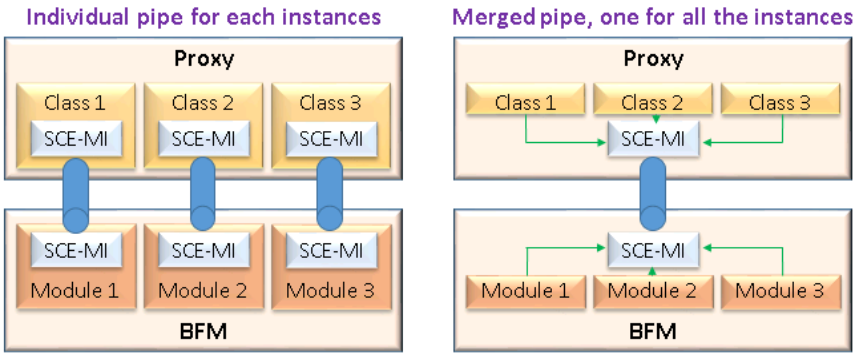

Figure 7: TB Architecture with merged SCE-MI pipe

- Always do synchronous access: Asynchronous access to the pipes created behavioral evals, so changes were made to access the SCE-MI pipes synchronously. It is done by setting the parameter of the pipe, IS_CLOCKED_INTF to one. This effectively allowed the pipe to operate based on clk, as shown in the figure 8.

```
//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 8: Synchronous SCE-MI pipe

- Optimized data transfer: Transferring small amount of data frequently had increased the behavioral evals by a significant amount. Instead of transferring "*N*" elements, with size of each element being a byte, the data was combined and sent as one element of size "*N*" bytes. This implementation effectively reduced SCE-MI access by *"N"* times thereby boosting the performance. A sample code snippet is shown in figure 9.

```
//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 9: Size of each element in a SCE-MI pipe

- Synchronization should not be frequent: Data transfer between proxy and BFM causes simulator and emulator to synchronize under the hood. Acceleration was increased by avoiding frequent data transfers. Instead of sending one element of size five bytes at regular intervals, twenty elements were packed together and this reduced 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 10: Maximum number of elements in a SCE-MI pipe

- A way to clear the pipe: For-loop was used to remove any residual data present in the pipe during reset or interrupt.

```
//Inefficient way of clearing the pipe using 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 11: Clearing a SCE-MI pipe, using for-loop

Instead of using for-loop to clear the pipe, synchronous procedural block was used to achieve the same functionality. A clock, faster than the operational clock, was generated and used it to clear the pipe synchronously. This avoided all the behavioral evals due to the for-loop. The end of message (eom) mechanism with this procedure was coupled to signal the end of the residual data.

```
//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 12: Synchronously clearing the SCE-MI pipe

C. Savaging Direct Programming Interface

Function-based interface is bundled with the ability to communicate between different languages using DPI. It also gives the liberty to write one's own functions in one language and invoke it from another. DMI also utilizes the DPIs to access the predefined APIs. Additionally DPIs gives us the flexibility to write one's own functions as per the required functionality. Figure 13 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 through 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 = svGetScopeFromName(tbpath);
   svSetScope(tbscp);
   reg bfm wr(reg data);
}
```

Figure 13: C-function invoking BFM's method

Defined C-function can be imported and invoked from the proxy using DPI as shown in the figure 14. This in turns execute 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 reg c = function void write bfm reg(bit [31:0] data wr);
class proxy driver;
   bit [31:0] data wr;
   task reconfig;
      write bfm reg(data wr);
   endtask
endclass
```

Figure 14: Proxy using DPI to invoke C-function

Method implemented in the BFM is exported using the inbuilt DPI, as shown in the figure 15, 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 15: BFM exports its method through DPI

The modified architecture of the test environment after implementing all of the above optimization approaches is shown in the figure 16. A combination of SCE-MI DMI, SCE-MI function based interface and SCE-MI pipe based interface was used to achieve the required functionality. This new architecture offered significant performance improvement and the same is discussed in the upcoming section.

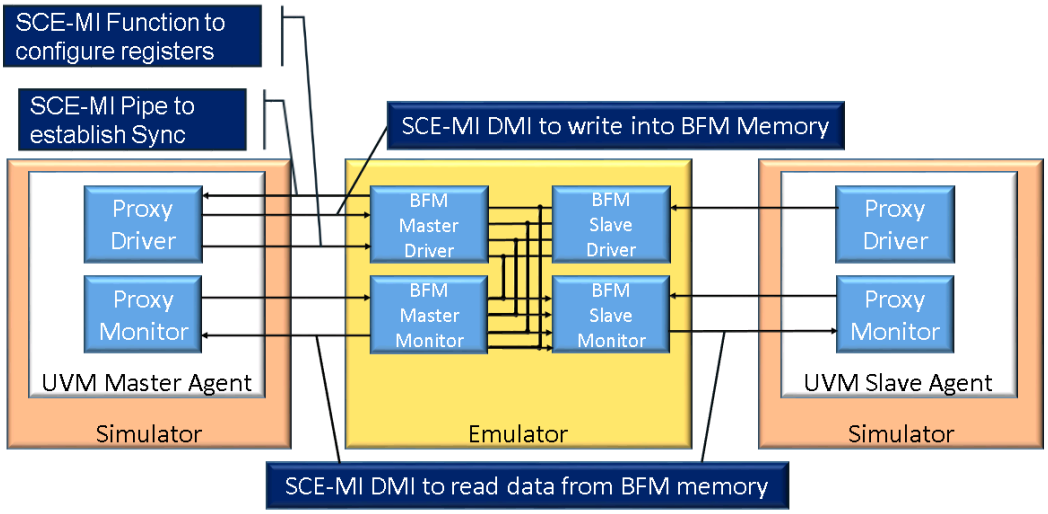

Figure 16: Optimized TB Architecture

V. RESULTS

The optimizations over the initial SCE-MI pipe usage, as discussed in this paper, has increased the acceleration performance by a significant margin. Table I shows the comparative improvement in the TBA performance before and after the optimizations. Along with the TBA run-time, Table I also compares behavioural evals and HW-SW syncs, which were the major performance degraders.

Table I. Comparative results of the implementation before and after optimization

Implementation	**TBA properties**			
	Gate count	***Bevals***	***HW-SW Sync***	***TBA Time***
Before optimization	~2M	18,299,173	4,728,998	~1hour
After optimization	~2M	218	18,439	5 min

The TBA runtime after the optimizations in SCE-MI usage effectively reduced by 92% in comparison with the non-optimized TBA run-time. Block level run, which took hours in simulation mode, came down to mere minutes in TBA mode. An acceleration of 40X was achieved using the TBA against the simulation mode, the run-time between the two modes are compared in the table II.

Table II. Comparison between simulation vs TBA runtime

Comparison	***Simulation time***	***TBA time***
Simulation v/s TBA	~6 hours	~9 min

VI. CONCLUSION AND FUTURE WORK

Modeling of an efficient TBA testbench demands efficient usage of SCE-MI. As shown in this paper HW-SW syncs and behavioral evals severely cripple the TBA performance. Choosing the appropriate SCE-MI flavors depending on the requirements and using them effectively eliminates the risk of accumulating HW-SW syncs and behavioral evals. In this work a combination of SCE-MI DMI, SCE-MI function based interface and SCE-MI pipe based interface was used to achieve the required functionality. As part of the future work, the optimized agents are planned to be integrated with the DUT and its behavior in the emulator will be analyzed. Furthermore, the agents 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] Hans van der Schoot and Ahmed Yehia, "UVM and Emulation: How to Get Your Ultimate Testbench Acceleration Speed-up", DVCon Europe - 2015