Efficient Testbench Architectures for SoC Designs using SystemC and SystemVerilog

Ashutosh Godbole
Infineon Technologies AG
Munich, Germany
ashutosh.godbole@infineon.com

Ingo Kuehn
AMD Saxony LLC & Co. KG
Dresden, Germany
ingo.kuehn@amd.com

Thomas Berndt
AMD Saxony LLC & Co. KG
Dresden, Germany
thomas.berndt@amd.com

ABSTRACT

When designs get bigger and more complex, the high level verification languages like SystemC, e, or SystemVerilog are needed to accomplish the verification task of SoC designs. An open source language like SystemC along with its verification extension (SCV) provides high level constructs and data types, randomization features, and object oriented paradigm, which are needed to thoroughly verify RTL designs. But the price paid is long simulation times due to the co-simulation bottle-neck.

This paper proposes a mixed-language testbench architecture based on SystemC and SystemVerilog, which is a significant improvement over conventional HVL verification environment. It presents a real world example where the approach was deployed to improve performance of an existing PLI-based SystemC-Verilog co-simulation. It makes use of Synopsys VCS features like SystemVerilog - DPI (Direct Programming Interface) and Transaction Level Interface (TLI) to raise the SystemC-Verilog co-simulation interface to a higher level of abstraction (transaction level). This dramatically increases the overall simulation speed as the data between the two language domains are exchanged in terms of transactions instead of signals.

Furthermore, the proposed mixed-language verification environment also improves the overall verification methodology by reusing low level verification components developed in SystemVerilog or simple Verilog. This helps to reduce the valuable time and effort spent on verification.

Table of Contents

Table of Figures

1.0 Introduction

The verification environment to verify System-On-Chip designs developed at AMD, Dresden was modeled in SystemC, while Verilog was used for RTL design. This may be a common structure found in many companies. The co-simulation of SystemC environment and Verilog design was achieved using a portable and tool independent Verilog-PLI based co-simulation interface. This co-simulation interface was responsible for synchronizing the SystemC and Verilog simulators and exchanging data between the two language domains. Due to the PLI overhead, the simulations took quite a long time. The goal of this paper is to propose a mixed language testbench architecture that improved this SystemC-HDL co-simulation performance.

Next section of this paper will discuss the use of high level verification languages in contemporary functional verification. Section 3.0 will explain the co-simulation bottleneck which occurs in SystemC-HDL simulations. The proposed testbench architecture will be the main focus of section 4.0. The section 'Comparison Results' will demonstrate with the help of a real world example, the performance gain achieved by this testbench architecture.

2.0 SystemC and SystemVerilog: High Level Verification Languages

2.1 SoC verification

A key issue in SoC design is integration of silicon IPs (cores). Integration of IPs directly affects the complexity of SoC designs and also influences verification of the SoC. SoC verification becomes more complex because of the many different kinds of IPs on the chip. A verification plan must cover the verification of the individual cores as well as that of the overall SoC. Various SoC applications require unique external interface constraints, and the verification team should consider those constraints early on.

The verification of these multi-million gate SoCs is a daunting task. Clearly, hardware description languages like VHDL/Verilog are inadequate for verifying these complex designs within the available time frame. Hence the need for high-level verification languages! High level verification languages like e, SystemC (in conjunction with SCV), and SystemVerilog come with the complete arsenal necessary for verification of huge designs. These HVLs are equipped with following features, which make them effective for verification:

- Object oriented paradigm and modularity
- High level programming language features such as complex data types
- Transaction based verification
- Support for constrained random generation

2.2 SystemC

While standard SystemC can be used to perform basic verification of a design, the SystemC verification standard (SCV) improves its capability by providing features for transaction based verification, constrained and weighted randomization, exception handling and other verification tasks. SystemC is a flexible, object oriented architectural modeling language designed for modeling multiple abstraction levels, including TLM. SystemC is implemented as a C++ library that incorporates concurrency and notion of time in the traditional C++ framework.

Pure SystemC may be insufficient to verify complex designs, but the SystemC Verification library (SCV), which can be treated as the SystemC verification extension makes SystemC an excellent hardware verification language (HVL). It includes all the features for modern verification, such as transaction based verification approach, data introspection, constrained and weighted randomization for stimulus generation, etc.

2.3 SystemVerilog

SystemVerilog is a rather new language built upon the Verilog language and is useful for both hardware design and verification. SystemVerilog extends Verilog into the systems space and the verification space. It adds extended and new constructs to Verilog-2001 for a higher level of abstraction for modeling and verification. The language enhancements provide more concise constructs for hardware description, while still providing an easy route with existing tools into current hardware implementation flows. The enhancements also provide extensive support for directed and constrained-random testbench development, coverage driven verification, and assertion based verification.

2.4 Race between the two

Engineers with a software background often prefer C/C++ or SystemC as the TLM language, while other engineers coming from a hardware design background may prefer SystemVerilog for verification. SystemC extends the C++ scope towards hardware, while SystemVerilog extends the Verilog scope to object orientation and testbenches. Both languages support concepts such as signals, events, and interfaces and object oriented methodology.

SystemVerilog is built on the top of Verilog. Verilog simulators like VCS provide integrated support for SystemVerilog. No separate simulator is required. Hence there is no co-simulation overhead. This is the particular feature that could be exploited to improve simulation performance. This paper describes how the verification environment can be built using both SystemC and SystemVerilog. It demonstrates how we can gain from both the languages by making a proper language choice for a given abstraction level within the testbench architecture.

Both the languages have some great features that are extremely useful in verification and both have some drawbacks. It can not be proclaimed that one language is better than the other for all purposes. The ultimate goal is to increase verification productivity and reduce the valuable time spent in verification. This calls for a shift in overall methodology and the testbench architecture. In this paper, we also demonstrate how low level verification components can be reused and how simulation speeds can be increased by mixing SystemC and SystemVerilog for developing verification environments.

The design decision of choosing SystemC or SystemVerilog is seldom made from scratch. The decisions are often based upon the existing testbench infrastructure and verification IP that is already available. It is not feasible to discard the existing verification infrastructure developed in SystemC completely, and then redevelop everything in SystemVerilog. This was the scenario at AMD, Dresden. The verification infrastructure in SystemC was already available. The simulation performance had to be improved without discarding this legacy SystemC verification

IP and at the same time, by making use of SystemVerilog's close binding with the VCS simulator kernel.

3.0 The Co-simulation Bottleneck

3.1 A typical testbench architecture for a bridge design

Figure 3.1 shows a legacy testbench architecture used for a typical bridge design. For other DUTs this architecture can be adapted (e.g. for some co-processors, only one side of the testbench exists). The DUT is in Verilog and the components of verification environment are described in SystemC. This testbench architecture will be used as a case study throughout this paper.

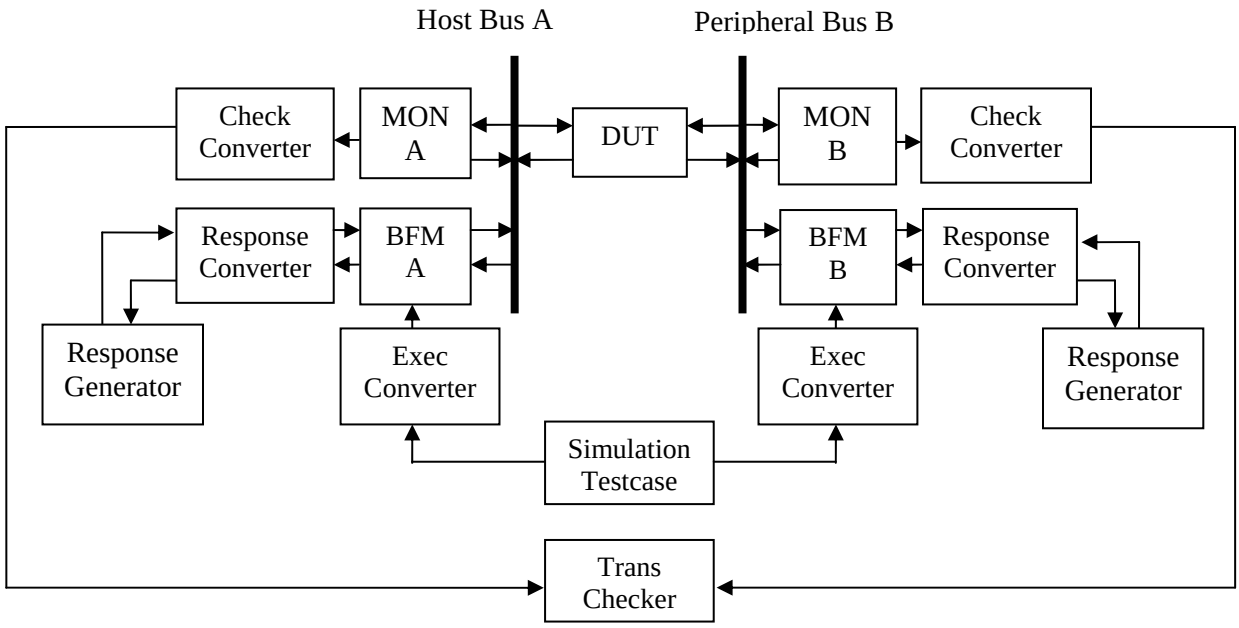

Figure 3.1 Testbench structure for a bridge design

3.2 A Verilog PLI-based SystemC-Verilog co-simulation interface

This section describes the legacy co-simulation framework used at AMD, Dresden. Co-simulation of the Verilog design and SystemC verification environment is carried out using a Verilog-PLI based co-simulation interface. This co-simulation interface is responsible for synchronizing the SystemC and Verilog simulators and exchanging data between the two language domains. Due to the PLI overhead, the simulations take quite a long time.

3.3 Exec path example

To delve deeper into the problem, let us consider the host transaction execution path (exec path). The host bus is AMD GeodeLink. On the device side we have LPC (Low Pin Count) bus. Figure 3.2 shows the DUT as a black box:

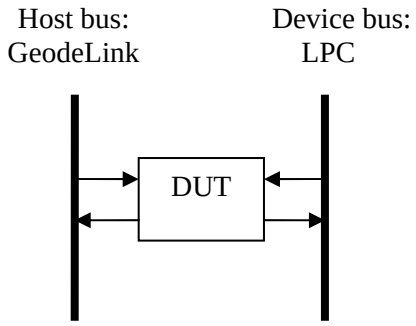

Figure 3.2 DUT block

The DUT acts as a bridge and converts the host bus transactions into LPC transactions. In the test case "lpc_traffic_tc", read/write transactions are applied on the host bus and LPC transactions are received on the LPC interface. The exec path refers to the testbench components, which are involved in generation and execution of the host transactions. Figure 3.3 shows the "exec" path:

The transaction flow through the exec path:

- The test case is the upper-most layer and it applies constraints to the transaction generator.
- The transaction generator then generates interface unspecific transactions obeying the imposed constraints. These transactions are called host transactions. The host transactions are then handed over to the exec converter.
- The exec converter translates the incoming request host transactions into interface specific transactions; GeodeLink transactions in this case.
- Finally the GeodeLink BFM receives transactions from the exec converter and converts those into relevant Verilog signaling on the interface. It is the only time consuming unit in the whole chain. Thus the BFM receives transactions and applies to the physical interface on the other side by spreading them in time.

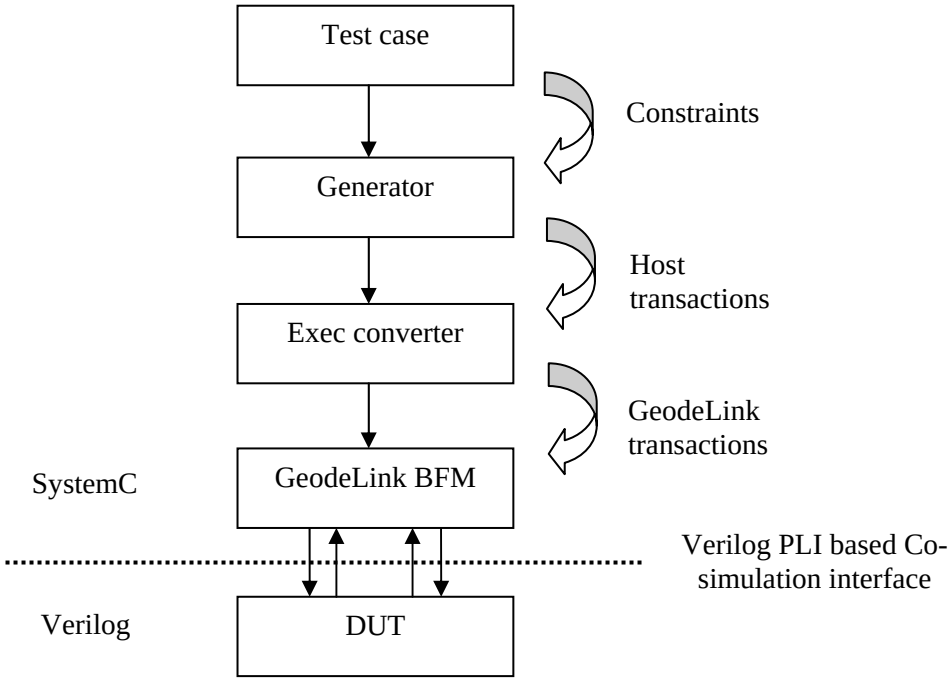

Figure 3.3 The exec path

3.4 PLI overhead

It can be seen in the figure 3.3 that the interface between Verilog and SystemC is below the GeodeLink BFM: at signal level. The event driven, PLI based co-simulation interface is responsible for the data exchange between SystemC and Verilog. Any PLI application is inherently slow and hence it significantly affects the co-simulation performance. More the bus activity more is the required SystemC-Verilog data exchange resulting into slow overall simulation.

4.0 Improving Simulation Performance

4.1 A DKI based solution

One of the solutions to the slow SystemC-Verilog co-simulation problem is to replace the existing PLI-based co-simulation interface with a co-simulation interface which does not have any PLI overhead. Synopsys VCS-Direct Kernel Interface (DKI) is such a signal level co-simulation interface between SystemC and Verilog. It is a high bandwidth solution, which boosts the simulation performance by reducing the PLI overhead. The extra protective PLI layer is removed and there is more direct interaction with the simulator's internal data structures; this results in high speed SystemC-HDL data exchange. Furthermore, as the VCS built-in SystemC simulator is used with DKI, the simulators are more tightly integrated. This also helps to improve the co-simulation performance to some extent.

Although DKI has a clear advantage over traditional PLI based co-simulation, it is still an optimization at signal level. The implementation details of co-simulation with DKI will not be

discussed in this paper, as the main focus of this paper is to go beyond the signal level co-simulation. The DKI approach is only included for comparing the approaches.

4.2 The DPI approach

Another solution to the slow SystemC-Verilog co-simulation problem is to raise the interface itself between SystemC and Verilog to a higher level of abstraction: from signal level to transaction level. When the interface is moved from signal level to transaction level, the co-simulation overhead will be reduced because the slower signal level SystemC-Verilog data exchange will get replaced by a faster transaction level data exchange.

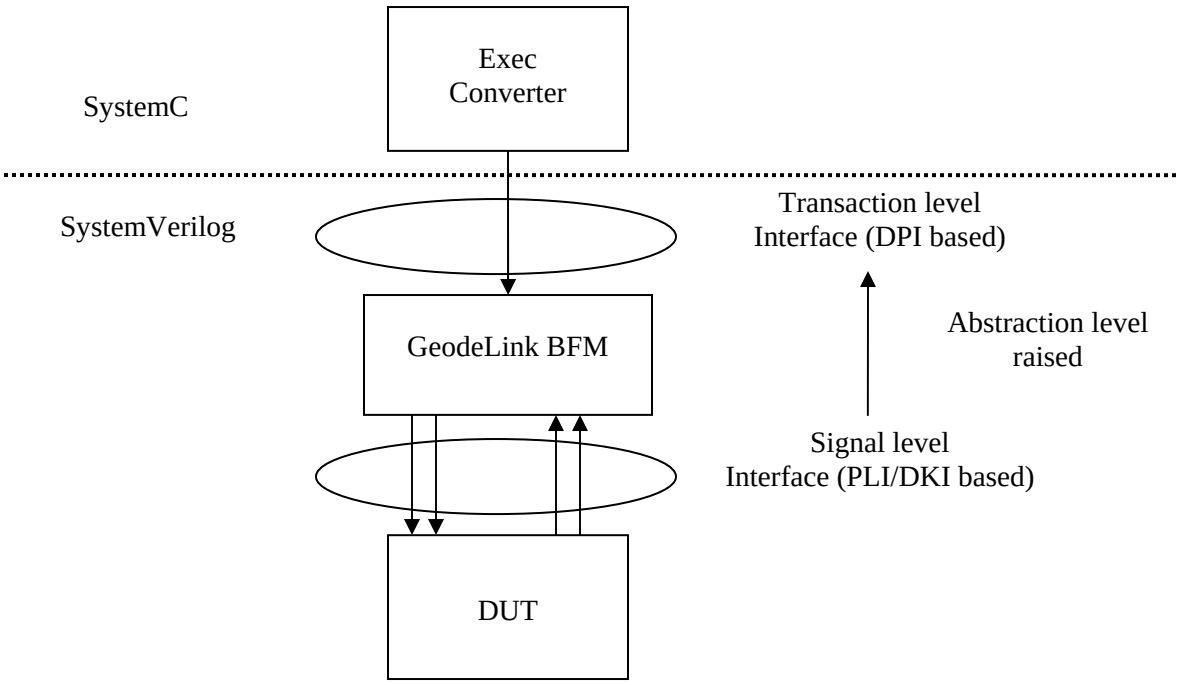

Figure 4.1 Transaction level interface between SystemC and SystemVerilog

4.3 Implementation details

To achieve this, the existing SystemC bus functional models (BFMs) and bus monitors were ported to SystemVerilog. These BFMs and monitors were connected to the higher layers of the verification environment using the SystemVerilog Direct Programming Interface (DPI). The transaction level data exchange between SystemC and SystemVerilog would take place through the SystemVerilog's direct programming interface (DPI). The other advantages of this approach are:

- The SystemVerilog components developed for simple low level testbenches and test cases could be reused.

- BFMs and monitors developed in SystemVerilog would be easier to debug for RTL designers.
- With this approach, the generators and checkers could be independently developed using higher level languages like SystemC / C++ while low level verification IP like BFMs and monitors could be developed in SystemVerilog / Verilog.
- As SystemVerilog integrates seamlessly into Verilog, the SystemVerilog – BFMs could be developed first and provide a primitive interface for directed stimulus. The random stimulus could be added later via SystemC. Work is not done twice as the BFMs are reused for the SystemC / C++ testbench.

4.3.1 Modified testbench architecture

Figure 4.2 shows the modified testbench architecture after implementing the BFMs and monitors in SystemVerilog. The architecture is partitioned in such a way that all the time consuming part is implemented in SystemVerilog, while all the non time consuming part is retained in SystemC.

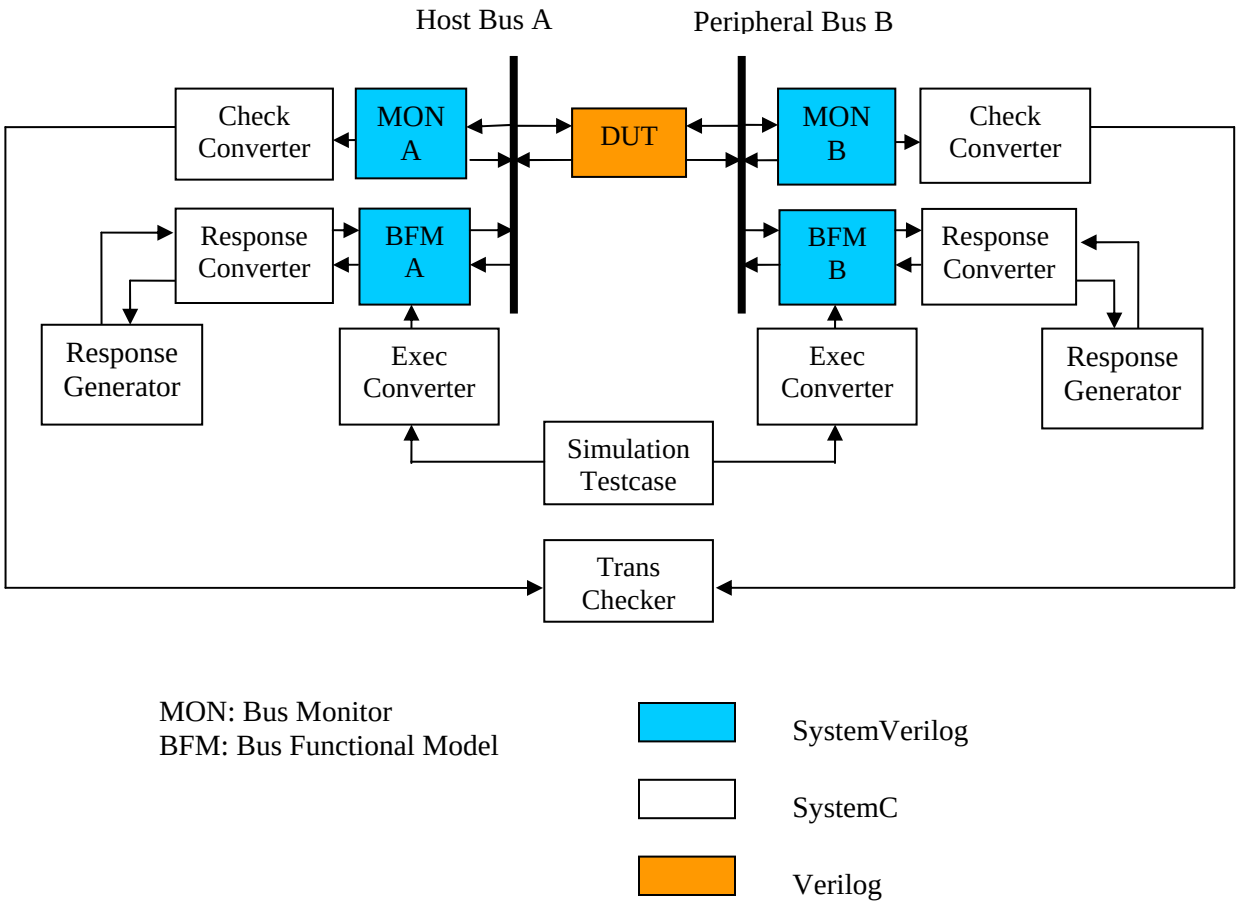

Figure 4.2 Modified testbench architecture

4.3.2 How to connect SystemVerilog module instance with SystemC/C++ object?

The SystemVerilog standard currently supports only C programming language as a foreign language layer. In principle, any foreign programming language with a C function protocol and linking model can be interfaced with SystemVerilog using DPI. Using DPI, SystemVerilog side can call C functions, but it does not have any knowledge of an elaborated SystemC module. Hence an extra layer was implemented which would connect a particular SystemVerilog module and a SystemC module.

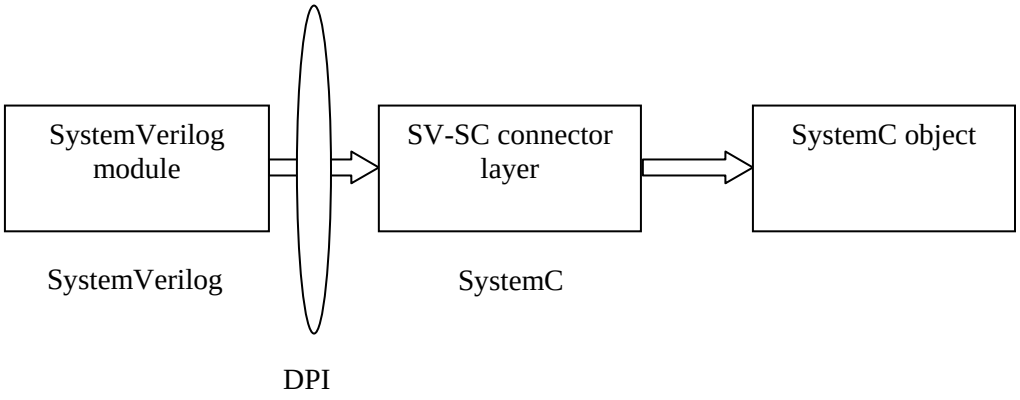

Figure 4.3 Connection of SV module instance with SC module

Figure 4.3 above shows this connecting layer between a SystemVerilog module and a SystemC module. This layer has a global C++ function, with an extern “C” directive. By default, all function types, function names, and variable names have a C++ language linkage. The extern "C" linkage specifier prevents the C++ compiler from mangling the name of a function and the function behaves as a normal C function. Such a function which follows C linkage mechanism can then be called from SystemVerilog side via DPI.

This C++ function without any name mangling would be called from a SystemVerilog module, which does not know anything about a SystemC/C++ object. This wrapper function would then retrieve a SystemC object from a static table of the corresponding class. Finally, the member function of the SystemC module (C++ object) would be called. This mechanism can be best understood with the help of an example. Consider an example of interfacing LPC monitor implemented in SystemVerilog with a checker implemented in SystemC/C++.

The following code snippet shows imported DPI function declaration in LPC SystemVerilog module:

```
program sr_lpc_mon_sv(

  lad,
  lclk,
  ldrq_n,
  lframe_n,
  lreset_n,
  serirq);

  `include "lpc_trans_sv.v"

  // Inputs
  input [3:0]      lad;
  input            lclk;
  input            ldrq_n;
  input            lframe_n;
  input            lreset_n;
  input            serirq;

  // send transaction to the checker via DPI
  import "DPI" context function void lpc_send_to_checker
  (input int unsigned cmd_dpi,     input int unsigned aspace_dpi,
   input int unsigned addr_dpi,    input int unsigned data_dpi,
   input int unsigned sync_dpi,    input int unsigned scnt_dpi,
   input int unsigned seq_num_dpi, input int unsigned trans_type_dpi,
   input int unsigned size_dpi,    input int unsigned start_stop_dpi
  );
  . . .
  . . .
```

The LPC transaction is then dispatched to the checker for checking. The SystemVerilog side calls the function “lpc_send_to_checker” from the SV-SC connecting layer. The SV-SC connector then forwards this call to the appropriate method of a checker object. The checker module is retrieved from a static table implemented in C++. On construction, the checker object has to be registered with this static table.

```
. . .
. . .

// required for DPI
#include "svdpi.h"

// function visible to SV lpc-mon
extern "C" void
lpc_send_to_checker(unsigned int cmd_dpi,  unsigned int aspace_dpi,
                    unsigned int addr_dpi, unsigned int data_dpi,
                    unsigned int sync_dpi, unsigned int scnt_dpi,
                    unsigned int seq_num_dpi, unsigned int trans_type_dpi,
                    unsigned int size_dpi, unsigned int start_stop_dpi
                   ) {

  // Local lpc transaction
  lpc_trans lt_local;
  // Local checker
  checker* checker_m;

  // Prepare transaction for the checker
  lt_local.set_originator(originator::create("lpc_mon"));
  lt_local.set_initial_delay(0);
  lt_local.set_cmd(sr::cmd_t(cmd_dpi));
  lt_local.set_addr_space(sr::addr_space_t(aspace_dpi));
  lt_local.set_dma(false);
  lt_local.set_addr(addr_dpi);
  lt_local.set_data(data_dpi);
  lt_local.set_sync(lpc::sync_t(sync_dpi));
  lt_local.set_sync_cnt(scnt_dpi);
  lt_local.set_sequence_num(seq_num_dpi);
  lt_local.set_tkind(trans::tkind_t(trans_type_dpi));
  lt_local.set_size(lpc::size_t(size_dpi));
  lt_local.set_start_stop(lpc::start_stop_t(start_stop_dpi));

  // retrieve the checker object from the static table
  checker_m = checker::get_checker_object("CHECKER1");
  sr_assertm(checker_m, "Checker must be set!");

  // send transaction to the checker
  checker_m->check(&lt_local);

}
```

Above code snippets explain how to connect a SystemVerilog module instance with a SystemC module (C++ object). In similar fashion, connections are made on the host bus side.

4.3.3 Reuse of low level Verilog BFMs/Monitors

Apart from simulation speed improvement, the other advantages of developing BFMs and monitors in SystemVerilog are:

(1) The low level verification IP like BFMs and monitors developed in Verilog for simple block level test cases could be reused.
(2) As SystemVerilog integrates seamlessly into Verilog, the SystemVerilog – BFMs could be developed first and provide a primitive interface for directed stimulus. The random stimulus could be added later via SystemC. Work is not done twice as the BFMs are reused for the SystemC / C++ testbench.

While porting SystemC BFMs to SystemVerilog, this reuse philosophy was utilized. Low level Verilog tasks for driving the host bus were available. The SystemVerilog host bus BFM was developed on top of these tasks, thus reusing the tasks. All the transactions received from SystemC side are first queued in the SystemVerilog host bus BFM. Then the main polling thread calls an appropriate low level task to apply the transaction over the host bus. The queue decouples the time consuming part of the BFM from the DPI interface. Hence the DPI calls on the SystemC side (exec interface) are simple non-blocking function calls. No special synchronization of simulators is required. Figure 4.4 explains the interfacing of SystemVerilog BFM and the higher layers of SystemC verification environment.

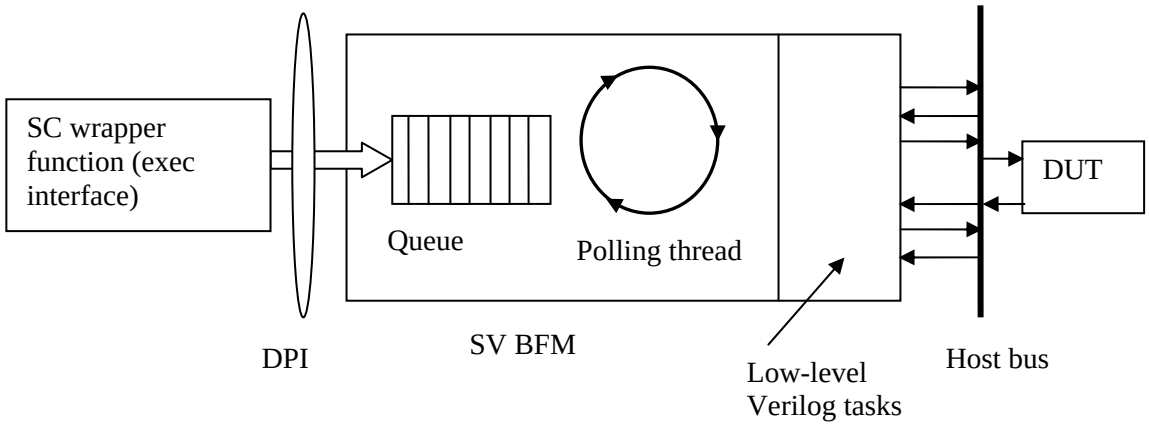

Figure 4.4 Reuse of low level Verilog BFM tasks

The following code snippet demonstrates this low level Verilog BFM reuse:

```
. . .
// Transaction polling thread
initial begin
  while(1) begin
    @ (posedge mb_clk or mb_reset);
    if (mb_reset) begin
    end
    else begin
      if (GeodeLink_trans_queue.size() != 0) begin

        // get the next transaction from the queue
        mt_curr = GeodeLink_trans_queue.pop_front();

        mb_asmi = mt_curr.get_asmi();
        mb_err  = mt_curr.get_mb_error();

        // call appropriate low level task to execute the transaction
        case (mt_curr.get_rqst_type())
        mb_coh_read: begin
          mem_read(mt_curr.get_rqst_type(), mt_curr.get_rqst_ad(),
                   mt_curr.get_rqst_sz());
        end
        mb_non_coh_read: begin
          mem_read(mt_curr.get_rqst_type(), mt_curr.get_rqst_ad(),
                   mt_curr.get_rqst_sz());
        end
. . .
. . .
        endcase
      end // if (GeodeLink_trans_queue.size() != 0)
    end // else: !if(mb_reset)
  end // while (1)
end // initial begin
```

4.3.4 Implementation challenges

The low level Verilog BFMs can not be directly reused without any modifications due to following reasons:

(1) Verilog features like ‘assign’ statements, always blocks, modules are **not** allowed within SystemVerilog program blocks.
(2) SystemVerilog high level language features like classes, associative arrays and queues are allowed **only** within program blocks.

Another major implementation challenge faced is - Synopsys support for various SystemVerilog features. Not all SystemVerilog language features were supported by Synopsys VCS at the time of implementing this example. For example, features like operator overloading, associative array with user defined type keys, passing of unpacked structures through DPI, etc. were not

supported. This made the porting of SystemC BFMs and monitors to SystemVerilog difficult. With some workarounds, these problems could be overcome.

4.3.5 Implementation using VCS-TLI

For the sake of completeness, above concept was also implemented using the latest feature of Synopsys VCS called as VCS-TLI (Transaction Level Interface between SystemVerilog and SystemC). But since TLI is built on the top of DPI, the simulation performance achieved using TLI was same as in DPI approach. In our case, DPI was more attractive because being a part of SystemVerilog standard it provided more portability and tool independence. Moreover, VCS-TLI was not officially released at the time of implementing this example. This section gives a brief introduction of VCS-TLI.

VCS-TLI is built on the top of DPI. It hides the DPI from the user and makes things easier for the user who wants to connect SystemVerilog and SystemC modules at transaction level. Some of the manual work, which was described in the previous chapter, is reduced by using VCS-TLI feature. It acts as a wrapper for DPI and automatically generates SystemC and SystemVerilog source codes for the TLI adapters.

TLI enables the user to:

- Call interface methods of SystemC interfaces from SystemVerilog
- Call tasks or functions of SystemVerilog interfaces from SystemC

Methods/tasks can be blocking as well as non-blocking. Blocking implies that the call may not return immediately. The caller's execution is resumed exactly at the simulation time when the callee returns, so a blocking call consumes the same amount of time in both simulator domains. Non-blocking calls always return immediately. VCS-TLI also helps in synchronizing the two simulators in case of blocking calls, which is not possible with DPI. But for simple non-blocking function calls between the two language domains, DPI should suffice.

The use model of the transaction level interface consists of defining the interface by means of an interface definition file, calling a code generator to create the TLI adapters for each domain, and finally instantiation and binding of the adapters.

Figure 4.5 illustrates how the VCS-TLI works for the case of ‘SystemVerilog calling SystemC interface functions’. It hides the DPI from user and uses it as an underlying mechanism for data transfer between the two language domains. The automatically generated wrappers/adapters forward call to SystemC interface method and take care of synchronization between SystemC and SystemVerilog.

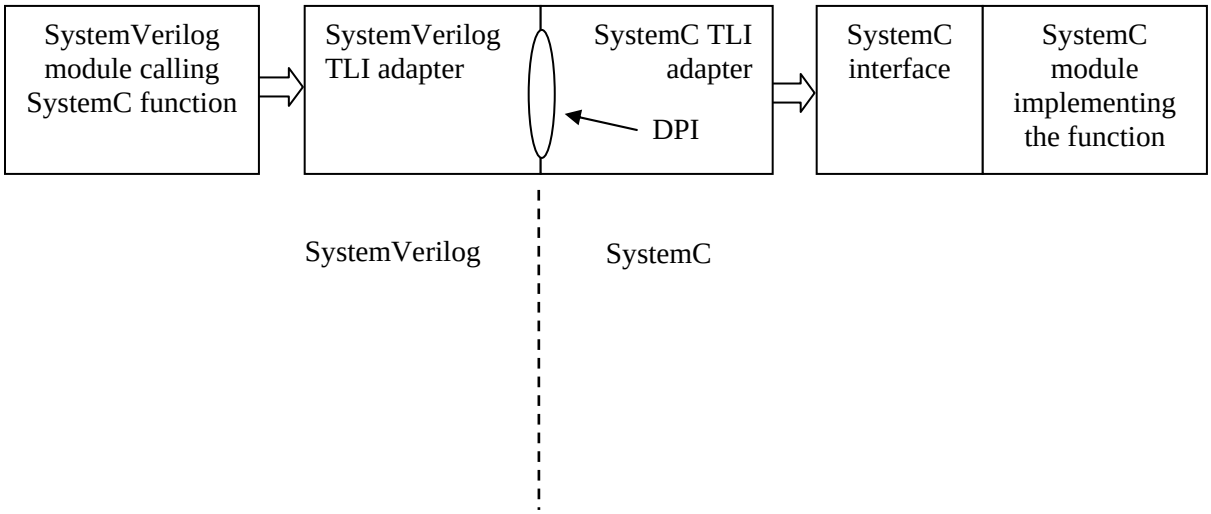

Figure 4.5 VCS-TLI usage model (SV calling SC)

5.0 Comparison Results:

5.1 Effect of different simulation run lengths

In this section, the approaches are compared for different simulation run lengths. The transaction count starts at 10 and then it is increased from 50 to 1600, doubling the number of transactions in every simulation. The UNIX/Linux 'time' command is used for measuring CPU time. The simulation times (CPU time) are noted for all the approaches and the trends are plotted as follows:

5.1.1 Average CPU time (in sec) Vs Number of transactions

Transaction Count / Approaches	10	50	100	200	400	800	1600
PLI Cosim	6.43	10.39	15.42	24.20	43.09	80.48	161.83
DKI	4.97	8.38	12.92	20.38	36.82	68.95	140.05
DPI	5.52	7.92	10.98	15.81	27.37	49.89	101.03

Table 5.1 Average CPU time (in sec) Vs Number of transactions

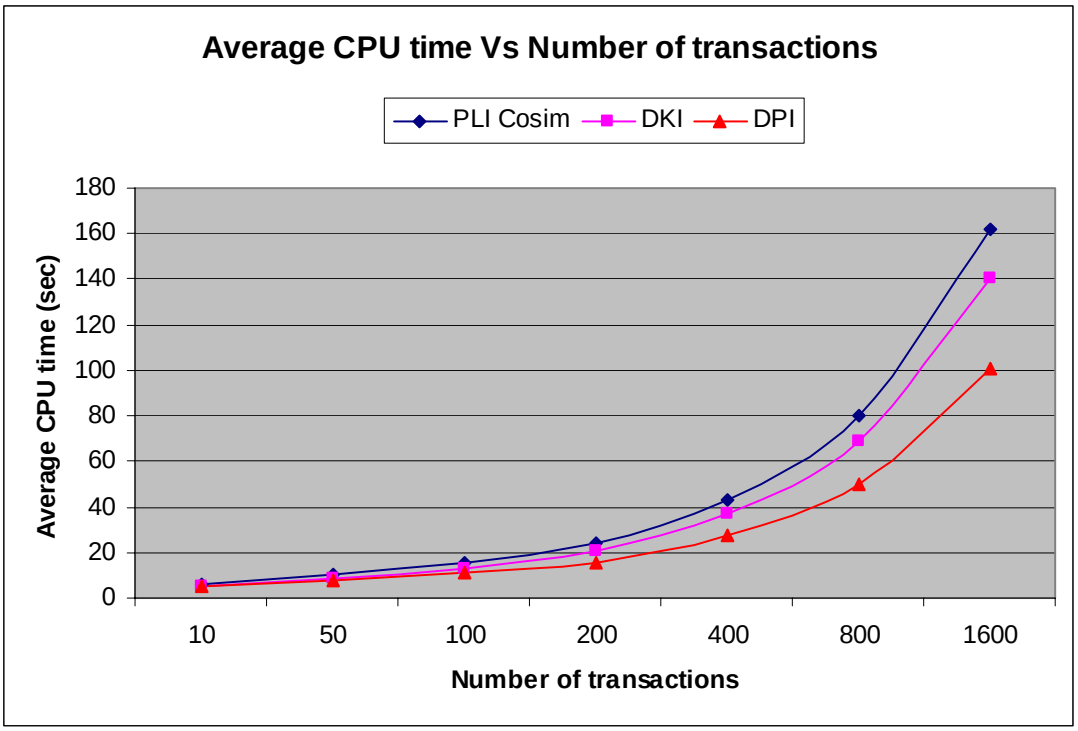

Figure 5.1 Graph of Average CPU time (in sec) Vs Number of transactions

In the above graph, it can be seen that the time required for co-simulations using the DKI and DPI is always less than the PLI based co-simulation, even if the transaction count is increased. This justifies the advantage of DKI and DPI based co-simulation strategies.

From the above data, the relative performance improvement of DPI approach can be analyzed by calculating the percentage speed increase of the approaches over PLI based co-simulation.

5.1.2 Percentage speed increase over PLI-based co-simulation

Transaction Count / Approaches	10	50	100	200	400	800	1600
DPI	15.21%	31.81%	41.00%	53.63%	56.26%	57.53%	58.16%

Table 5.2 Percentage speed increase over PLI-based co-simulation

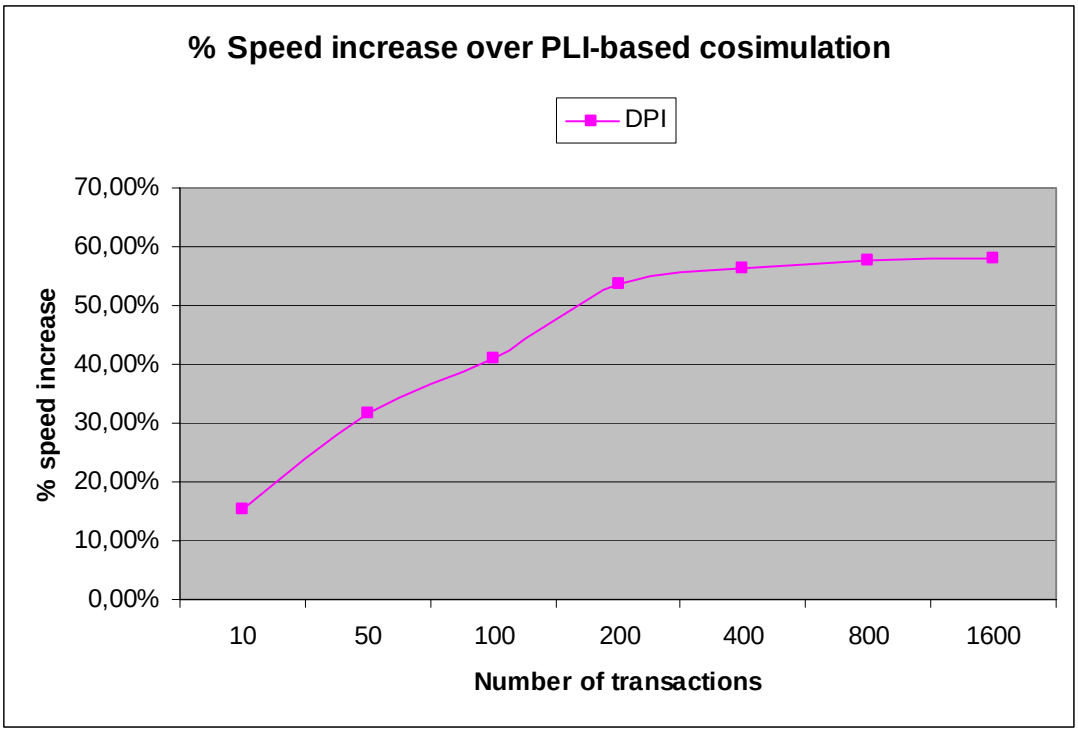

Figure 5.2 Percentage speed increase over PLI-based co-simulation

From the above graph, it can be observed that the relative speed improvement stabilizes roughly after 200 transactions. The percentage speed improvement over PLI-based co-simulation is fairly constant after this point. The changing nature of the curves during the 'zoomed-in' initial phase can be attributed to the initialization activities taking place during the co-simulation program execution, e.g. loading of runtime libraries.

6.0 Conclusions

The main objective of this paper was to present an efficient testbench architecture, which improves simulation performance. To justify the concepts, the approaches were tried out on real world SoC design blocks. The results achieved illustrate how this test bench architecture significantly improves co-simulation performance and the overall verification productivity.

In order to verify huge and complex System-On-Chip designs within the available timeframe, we need a high level verification language like SystemC. But verification of a Verilog design with a SystemC environment comes with an inherent problem of slow co-simulation. The legacy SystemC-Verilog co-simulation interface at AMD was Verilog PLI based which slowed down the overall simulation speed. This paper presented the ways to circumvent this problem.

The first probable solution presented was to replace the PLI-based co-simulation interface with an interface, which does not have any PLI overhead. The VCS Direct Kernel Interface (DKI) – a SystemC-HDL co-simulation interface from Synopsys Inc. could be the solution.

The second approach was to raise the interface between SystemC and SystemVerilog to a higher abstraction level. By adopting this approach, overall simulation speed increases because the slower signal level SystemC-Verilog data exchange gets replaced by a faster transaction level data exchange. To prove this concept, the existing SystemC BFMs and monitors were first ported to SystemVerilog, which were then connected with the upper layers of existing SystemC verification environment by SystemVerilog Direct Programming Interface (DPI). It was also shown how low level verification components could be reused, which is another step towards methodology improvement.

The paper also introduced the new Synopsys VCS feature called Transaction Level Interface (TLI). The TLI is built on the top of DPI and essentially hides the DPI from the user. The advantage it provides over DPI is - some reduction of manual effort. It automatically generates the adapters required for connecting a SystemVerilog module instance with a SystemC module.

Finally, all the approaches were compared on the basis of simulation time gain for different simulation run lengths. The decision to choose any approach is governed by several factors. There is an apparent tradeoff between performance and development effort. Legacy testbench structure and the overall existing flow also play a major role in making this informed decision.

To sum up, this paper demonstrated how we could gain from SystemC as well as SystemVerilog by making a proper language choice for a given abstraction level within the test bench architecture. Embracing just one language or technology may not be very beneficial. An integrated environment leveraging the advantages of both SystemC and SystemVerilog can be more effective.

7.0 Acknowledgements

I express my sincere gratitude to my mentors Ingo Kuehn and Thomas Berndt for their valuable advice, guidance and time. I especially thank Ingo Kuehn for reviewing my thesis and encouraging me whenever I faced any mental block. I also enjoyed all the technical discussions we had pertaining to my thesis.

I would also like to thank Matthias Leonhardt, Manager Design Verification, for giving me the opportunity and resources to carry out my thesis work at AMD, Dresden.

I am thankful to Frank Winkler and Dirk Huthmann from AMD, Dresden for all the technical help they provided during the course of my thesis. I take this opportunity to thank Ulrich Holtmann, Angshuman Saha and Dinh Kim Bui of Synopsys, Inc. for their support in solving some Synopsys tool related issues.

8.0 References

[1] SystemVerilog 3.1a Language Reference Manual
[2] Sauber SystemC Verification Environment documentation (AMD internal documentation)
[3] Synopsys VCS/VCSi User Guide, Ver X-2005.06-SP1, Ver Y-2006.06
[4] SystemC 2.0.1 Language Reference Manual, Rev 1.0
[5] *Rindert Schutten, Janick Bergeron (Synopsys, Inc.)*, Transaction-Level Modeling: SystemC and/or SystemVerilog, Synopsys Verification Avenue Technical Bulletin, Vol. 6, Issue 1, March 2006
[6] *Stuart Sutherland (Sutherland HDL, Inc.)*, Integrating SystemC Models with Verilog and SystemVerilog Models Using the SystemVerilog Direct Programming Interface, SNUG Europe 2004
[7] *Ulrich Holtmann (Synopsys, Inc.)*, Transaction Level Modeling: Integrated SystemC – SystemVerilog environment, SNUG Europe 2006