"C" you on the faster side: Accelerating SV DPI based co-simulation

Parag Goel,
Synopsys, India
91.80.40188000
paragg@synopsys.com

Amit Sharma,
Synopsys, India
+91.80.40188000
amits@synopsys.com

Hari Vinodh Balisetty
Broadcom,
+1.408.821.0362
harivino@broadcom.com

Abstract: The SystemVerilog Direct Programming Interface (DPI) based infrastructure is now increasingly used in software simulations to mimic real time HW SW interaction in SoC's. It is also used to integrate C/C++ based architectural models and also to create C/C++ based tests which can later be reused in post silicon validation as well as in simulation acceleration using emulation based platforms. C/C++ based testbenches are now frequently used for transactor based emulation and SOC validation. The requirements for complex protocols such as PCIe require the C/C++ components to be able to handle concurrency. However, for verification engineer conversant primarily with HDLs and HVLs such as SystemVerilog, non-optimal usage of the SV DPI based infrastructure can lead to major inefficiencies. This paper will focus on the techniques that would enable us to ensure that the communication overhead is minimal between the software and hardware components. It will also talk about how to leverage multi-threading for C/C++ based threads to meet the requirements of controlling hardware components. Additionally, it will lay down guidelines for efficient usage of C/C++ in the context of verification requirements.

Categories and Subject Descriptors
Co-simulation, Simulation Acceleration, Transaction Level Modeling
General Terms-Verification, Methodology, Verification IP
Keywords—SystemVerilog, Direct Programming Interface

I. Introduction

With an increase in software content in today's SOC, it becomes imperative to model hardware software interaction in simulations. Additionally, C/C++ models are typically used as golden reference models during simulation. C/C++ testbenches which can be reused in emulation and post silicon validation has also become common use case for reusability. The SystemVerilog Direct Programming Interface (DPI) was developed to standardize the interface between SystemVerilog and a foreign programming language that can be in C or C++. The SV DPI allows SystemVerilog to call a C function or task just like any other native SystemVerilog function/task Variables passed directly to/from C/C++. This greatly simplifies the usage of C/C++ components in a HVL testbench. This paper will focus on the techniques that would enable us to ensure that the communication overhead is minimal between the software and hardware components when using SystemVerilog DPI. From the perspective of software/testbench code running on the host, the paper will put forward recommendations for ensuring schemes are adopted which ensures that such execution will not negatively impact the overall performance of an emulated setup.

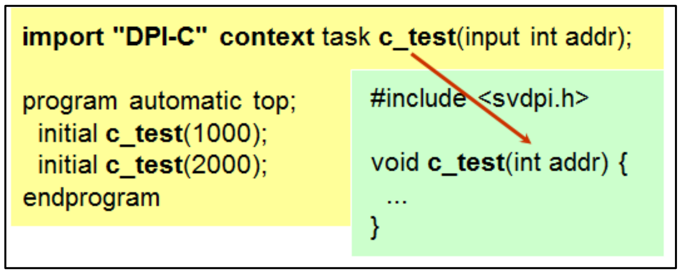
II. SystemVerilog DPI: Key Aspects

The SV DPI provides an interface between SystemVerilog and a foreign programming language. This direct interface can significantly reduce the complexity of interfacing testbenches to C/C++ models. It thus provides a more practical mechanism for connecting C/C++ code to SV without knowledge and overhead of VPI/PLI.

For DPI based communication, there are two major classifications that exist based on the direction of communication or rather where the definition of the task/function lies (SystemVerilog or C/C++).

'Import' Tasks/Functions

These are the methods whose definitions exist in C/C++. These methods are invoked from the SystemVerilog code. Imported tasks and functions can have zero or more formal input, output and inout arguments. However, arguments cannot be passed by reference. Imported functions can return a result or be defined as void function. The following is an example snippet of an 'import' function.



```
import "DPI-C" context task c_test(input int addr);

program automatic top;
  initial c_test(1000);
  initial c_test(2000);
endprogram
```

```
#include <svdpi.h>

void c_test(int addr) {
  ...
}
```

Again the 'imported' function itself can be declared as "context" or "pure" and imported task can be declared as "context". A function can be specified as *pure* when its result depends solely on the values of its input arguments. A pure function is assumed not to directly or indirectly (i.e., by calling other functions) perform the following:

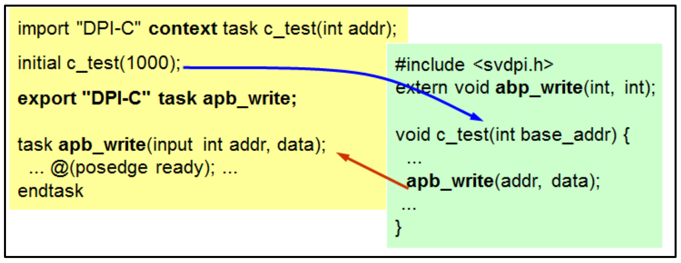
a. Perform any read or write operations from or to files
b. Access environment variables, objects from the operating system or other processes, shared memory, sockets, etc.
c. Access any persistent data, like global or static variables

By default import tasks and functions are non-context. They shall not access any data objects from SystemVerilog other than its actual arguments and thus they do not hinder simulator optimizations. However, there might be a need for an 'import' function to access or modify simulator data structures via the Programming Language Interface (PLI) or Verilog Procedural Interface (VPI) calls or making a call back into SystemVerilog through an 'export' method. Such tasks and functions would then be 'context' tasks/functions.

The effects of calling PLI or VPI functions or SV tasks/functions can be unpredictable; Such calls will fail if the caller requires a context and if that not been properly set.

Export Tasks/Functions

In case of Export tasks or functions, the definition of the task or function is in SystemVerilog and is invoked from the C/C++ domain. Functions/tasks which are members of classes cannot be exported, but all other SystemVerilog functions/task can be exported. Exported functions/tasks are always *context*. The following is an example snippet of an 'export' function.



```
import "DPI-C" context task c_test(int addr);
initial c_test(1000);
export "DPI-C" task apb_write;

task apb_write(input int addr, data);
  ... @(posedge ready); ...
endtask
```

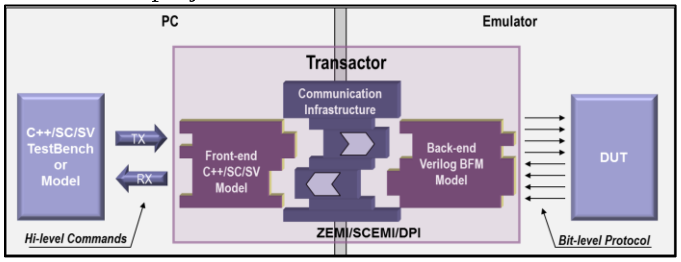
```
#include <svdpi.h>
extern void abp_write(int, int);

void c_test(int base_addr) {
  ...
  apb_write(addr, data);
  ...
}
```

It is illegal to call an exported task from an imported function. It is legal for an imported task to call an exported task only if the imported task is declared with the **context** property.

III. Co-simulation Using DPI

Transactor Based Verification

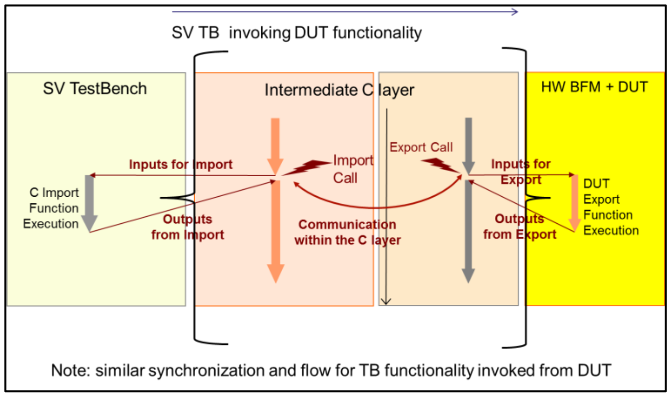
Transaction-based verification changes the level of abstraction at which your testbenches are written. Instead of writing testbenches based on signal assignments and signal monitoring, the testbenches are written with high-level commands or actions (such as *read something* or *get something*). Using these high-level commands, testbenches are easier to write and faster to create. A transactor is a behavioral model that gives a higher representation of an interface. This interface can be simple or complex, standard or proprietary. For each behavior, a transactor can be written once and reused as many times as required in one or more projects.


PC
Emulator
Transactor
Communication Infrastructure
C++/SC/SV TestBench or Model
TX
RX
Front-end C++/SC/SV Model
Back-end Verilog BFM Model
DUT
ZEMI/SCEMI/DPI
Hi-level Commands
Bit-level Protocol

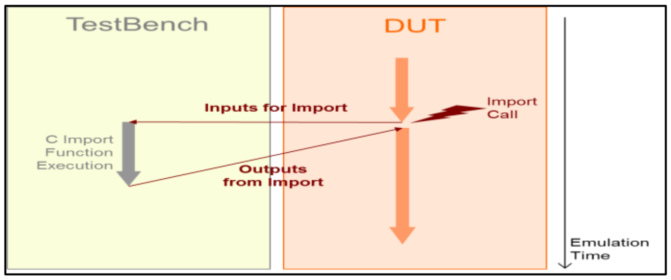
Thus it is a bridge from untimed transaction-based testbenches to the signal-level input to the DUT. They are typically used for Host-to-DUT SoC bus-level structures, standard communication protocols and peripheral interface standards. From a co-simulation context, the testbench and emulated DUT are synchronized only when required and thus the testbench and DUT can run in parallel and transactions can be queued. The speed improvement over cycle-based can be orders of magnitude faster reaching tens of MHz

Usage of DPI in the Context of transactors

When it comes to co-emulation, a simulator and an emulator in conjunction can use DPI to create the transactor bridge such that each function call is a transaction that is automatically off-loaded into the emulation hardware.

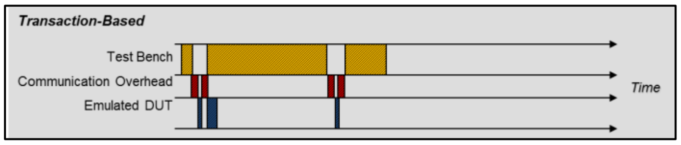

SV TB invoking DUT functionality
SV TestBench
Intermediate C layer
HW BFM + DUT
Inputs for Import
Import Call
Export Call
Inputs for Export
C Import Function Execution
Outputs from Import
Communication within the C layer
Outputs from Export
DUT Export Function Execution
Note: similar synchronization and flow for TB functionality invoked from DUT

In simple terms, for a SV/Verilog testbench running on the host simulator, we have an ‘import’ task/function which carries the relevant data to the hardware side through the task/function arguments. The Intermediate C layer then does minor data processing and sends the data and invokes the hardware time consuming method through an export task. Similar, communication in the other direction is achieved when the DUT running in the hardware needs to communicate with the software testbench. In specific context, when we have a synthesizable testbench, we might just need to have some processing done on the host side (When hardware needs a complex operation result which is easier to implement in software) then the usage context with DPI is made more simpler as shown below.



IV. Recommended usage for faster co-simulation performance

As seen below, in transaction-based co-emulation, the testbench and Emulated DUT are synchronized only when required whereas in traditional verification, the testbench interacts with the DUT at pin-level using PLI and a lot of traffic is going on, very often on each clock cycle.



Hence, the overall performance is dictated by the time spent in testbench and the communication overhead due to the DPI calls across the testbench and the emulated DUT. Hence, inefficiencies in the DPI communication layer itself can make a significant difference to the co-emulation performance. Inefficient communication especially if bulk transfers are involved can significantly slow down simulation performance.

The inefficiencies can also arise due to incorrect data types used, inappropriate handling of dynamic data structures, the size of the transfers itself and redundant exchange of data. An incorrect usage of formal/actual arguments can lead to memory leaks in the interface which can be quite difficult to debug.

Frequency of DPI calls

To ensure that the testbench does not slow down the overall performance, it would be prudent to use multi-cycle transactions. If DPI calls are very frequent, the recommendation is to combine them and prune argument lists to reduce the amount of data transferred from the simulator:

1. Each DPI call in hardware results in a transfer of an n-bit packet header word. Additional n-bit words are added to the packet to convey argument values. For example, if both f() and g() are functions that take a one bit wide argument each (SV 'bit' data type), then "f(a); f(b); g(c); g(d);" causes a transfer of 4*2(n-bit) words.
2. One way to minimize the total amount of words transferred is to combine calls. In the above example, by creating a new wrapper function ffgg(bit a, bit b, bit c, bit d), we can reduce the total transfer size from 4*2(n-bit) words to2(n-bit).

3. Make sure that each argument is declared to be as narrow as possible -- for example, never use the 'int' data type if the argument can only be 0 or 1 in case of (hardware) emulation import DPI calls.
4. When passing a limited number of constants, try to create multiple versions of DPI functions to eliminate the need for constants -- but only do it if this reduces the total number of words to be transferred. For example, we have a method which has index as one of the arguments. This method if called multiple times with different values of index, say 1, 2 etc. like,
5. compare_values (index, x, y) then it would be better to have multiple methods instead, compare_values_1(val,addr), compare_values_2(val,addr) etc.

Enabling efficient communication

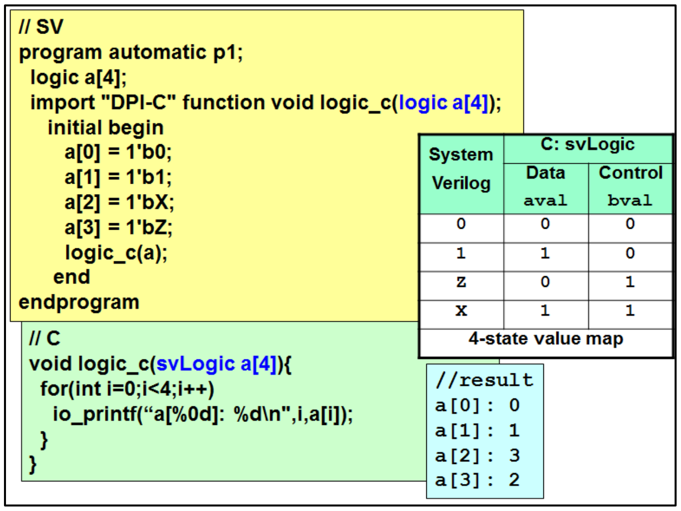
Appropriate usage of language constructs

1. Declare imported functions as 'pure' whenever possible, to allow for more optimizations: Calls to pure functions can be targeted by generic simulator optimizations or replaced with the values previously computed for the same values of the input arguments.

2. Avoid using large bandwidth if not required: Relevant data should be sent and additional variables not used by the DUT should be discarded.

3. Using the correct data types: While invoking DPI functions from the testbench, users should preferably use data types that map to native C-data types, like char, shortint etc. Avoid usage of scalar data types (bit/reg/logic) and the packed counterparts in case of (software) simulation of DPI calls. Also avoid using packed types like, arrays, structures, unions and enumerated data types. The scalar bit/logic/reg mapped to scalar "unsigned char", while the packed bit/logic/reg mapped to canonical form. So there is an additional conversion which may cause performance to degrade. The SystemVerilog-specific types, including packed types (arrays, structures, unions), which have no natural correspondence in C. For these the designers can choose the layout and representation that best suits their simulation performance. The representation of data types such as packed bit and logic arrays are implementation-dependent, therefore applications using them are not binary-compatible (i.e. an application compiled for a given platform will not work with every SystemVerilog simulator on that platform).Every packed type is eventually equivalent to a packed one-dimensional array. On the foreign language side of the DPI, all packed types are perceived as packed one-dimensional arrays regardless of their declaration in the SystemVerilog code. Open array in SV side is always mapped to svOpenArrayHandle in C side: typedef void* svOpenArrayHandle

4. Avoid using the 4-state passing across the C interface.



```
// SV
program automatic p1;
  logic a[4];
  import "DPI-C" function void logic_c(logic a[4]);
    initial begin
      a[0] = 1'b0;
      a[1] = 1'b1;
      a[2] = 1'bX;
      a[3] = 1'bZ;
      logic_c(a);
    end
endprogram
```

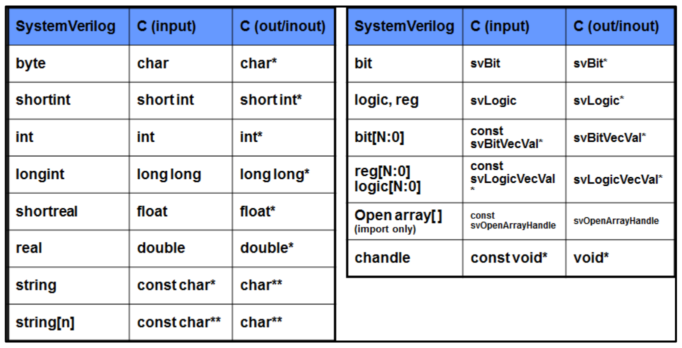
```
// C
void logic_c(svLogic a[4]){
  for(int i=0;i<4;i++)
    io_printf("a[%0d]: %d\n",i,a[i]);
  }
}
```

System Verilog	C: svLogic	
	Data aval	Control bval
0	0	0
1	1	0
z	0	1
x	1	1

4-state value map

```
//result
a[0]: 0
a[1]: 1
a[2]: 3
a[3]: 2
```

5. Map the compatible types for arguments in import method definition and declaration for correct results. Each data type that is passed through the DPI-C requires two matching type definitions: SV and C definitions. It is the user's responsibility to correctly declare compatible data types. The DPI does not check for type compatibility



SystemVerilog	C (input)	C (out/inout)
byte	char	char*
shortint	short int	short int*
int	int	int*
longint	long long	long long*
shortreal	float	float*
real	double	double*
string	const char*	char**
string[n]	const char**	char**

SystemVerilog	C (input)	C (out/inout)
bit	svBit	svBit*
logic, reg	svLogic	svLogic*
bit[N:0]	const svBitVecVal*	svBitVecVal*
reg[N:0] logic[N:0]	const svLogicVecVal *	svLogicVecVal*
Open array[] (import only)	const svOpenArrayHandle	svOpenArrayHandle
chandle	const void*	void*

There are important differences in the ways a simulator handles a packed type versus an unpacked type. A packed array (either as a separate entity or an element of an unpacked array) is treated as a vector. A vector can be used as a single entity in an arithmetic operation (e.g., *tagPacked >= tagPacked + 1'b1;*) and all bits of a vector is guaranteed to be contiguous. This increases overall simulation performance.

Packed types of SystemVerilog do not have any natural counterpart in the C domain. Nor does SystemVerilog LRM provide any direction regarding how this would be done. This is because simulators often optimize packed types for performance and hence the exact details of how a packed type is treated are implementation dependent.

Recommendations for more efficient data transfers

1. Perform the processing where it makes sense: Data processing usually done in testbench. Control processing usually done in synthesized part of the testbench
2. Use the most narrow data type possible to transfer information to the host PC. Avoid unused bits.
3. Arguments widths <= 32 bits result in better performance.
4. If there are large numbers of very short vectors, it may be best to concatenate them in Verilog, and unpack on the C++ side. However, this will only lead to efficient code if the concatenation is aligned on a 32-bit boundary.
5. Sending vectors greater than 32 bits that aren't aligned on a word boundary will be one of the most inefficient transfer scenarios
6. 64-bit wide integers (longint) will also deliver good performance.
7. Wide bit vectors are good if we can align them (multiples of 32-bits will always be aligned).
8. Avoid logic vector (4-state) arguments, unpacked structures.

If the memory/array is 16-bit/8-bit wide then use appropriate data type for building C-side arrays. "*int*" is fine for very small (few hundred items) memories. Using wider elements than needed is a bad idea for two reasons: RAM usage goes up, and CPU cache may become clogged with junk.

Communication schemes

Offline processing

One technique for achieving it is that the parameters that are passed via the DPI call to the 'C' side are off-loaded for offline process by assigning these to some global data structures defined in the imported domain and the main function resumes the execution of the functionality further. Now the methods look at these global structures to do the required functionality in this case a data comparison. But one can achieve such functionality if and only if the import task/function doesn't have a return value or doesn't have output arguments. In case the processing results are required to be passed back to the calling domain then this technique may not actually work.

Efficient File I/O

It is also important to hand file I/O efficiently as in specific scenarios, it constitutes a major component in the overall elapsed time for co-emulation. This is especially true in the context of backdoor loading of HW memories required for SOC based environments. Some recommendations in this area are:

1. Usage of the in-built standard function like (std::ifstream file(file_path);) for reading the text file to be loaded increases processing speed.

```
void read_file(char* file_path, int* buffer) {
  printf("The file loading from path:%s\n",
file_path);

  ifstream file(file_path);
  int value = 0;
  while(file >> hex >> buffer[value++]) { }
  if(file.eof()){value--;}
  file.close();
}
```

2. Usage of the binary files to load instead of other formats like hex/octal etc. The loading of binary files have been observed to increase the

performance over 5X both in simulation and co-simulation.

Leveraging Multi-threading

C/C++ based testbenches are now frequently used for transactor based emulation and SOC validation. The requirements for complex protocols such as PCIe require the C/C++ components to be able to handle concurrency. For example, in the PCIe link training process, let’s say we have two software components for the Endpoint and the Root Complex. For the link training to occur both the DPI threads initiating the training process needs to be active concurrently. Since in a single threaded C based execution, only one thread can be active at any point in time, there would be a deadlock situation which can only be managed through a complex and non-optimal set of spawned out threads on the SV side. Multi-threading schemes when implemented on the software side helps speed up the processing on the ‘C’ side. For example, in case one needs to do multiple DPI calls at the same clock edge and these are mutually exclusive to each other. In such a scenario, one can actually create multiple threads in ‘C’ and get the processing done in parallel for these methods. One such example could be, file loading into software or hardware memories. In such situation if we have to pre-load the memories with specific data, we can actually fork off different threads for loading memories which can take place in parallel and loading of one memory doesn’t affect another. In one such case study, where we had to pre-load/initialize almost 56 huge memories even before triggering the execution in the hardware domain, we saw a significant improvement. Loading memories linearly was 7x slower than when done using multiple threads (210 seconds vs. 30 seconds)

```
pthread_t f[3];
printf("Loading started...\n");
  pthread_create(&f[0],  NULL, &load_file1,  NULL);
  pthread_create(&f[1],  NULL, &load_file2,  NULL);
  pthread_create(&f[2],  NULL, &load_file3,  NULL);

  pthread_join(f[0],  NULL);
  pthread_join(f[1],  NULL);
  pthread_join(f[2],  NULL);
printf("Loading done...\n");

void* load_file1(void*) {
  read_file("AA_Info.hex", PAT_MEM_AA);
  read_file("AB_Info.hex", PAT_MEM_AB);
}
void* load_file2(void*) {
  read_file ("BA_Info.hex", PAT_MEM_BA);
  read_file ("BB_Info.hex" , PAT_MEM_BB);
}
void* load_file3(void*) {
  read_file ("CA_Info.hex", PAT_MEM_CA);
  read_file ("CB_Info.hex" , PAT_MEM_CB);
}
```

Note: Multi-threading can also be a killer for performance when more threads than CPUs are used typically.

Enabling concurrency in the ‘C’ domain

Let’s take the case of the two PCIe testbenches with a user API layer in the C domain. These are connected back to back (one configured as Root Complex and other as Endpoint). It is required to perform the link training process, which essentially involves the simultaneous transactions from both ends. Here, we need to spawn out the link training process of the Root Complex and the Endpoint so that each model can execute and drive the interface and complete the link training process by mutually exchanging the required ordered sets.

Essentially to accomplish the training process, we need to introduce threads for both the modes RootComplex and EndPoint respectively.

```
void* rc_link_training(void* arg) {
  Gen3Pcie foo = *((Gen3Pcie*)(arg));
  while(!foo.pcie_rc->initBFM());
   ....
    foo.pcie_rc->runBFM(PCIE_GEN3::SeqRunUntilTrainingDone);
    while(!foo.pcie_rc->waitBFMStatusChange(foo._Status));
  ......
return (NULL);
}

void* ep_link_training(void* arg) {
  Gen3Pcie foo = *((Gen3Pcie*)(arg));
  if(foo.pcie_ep->initBFM())  {
   foo.pcie_ep->runBFM(PCIE_GEN3::SeqRunUntilTrainingDone);
  }
  return (NULL);
}
```

Then based on the mode of the BFM, we can create an appropriate ***pthread*** and for any number of instances of the BFM.

```
int XactorConfigure(int unsigned is_root, int unsigned inst_num) {
  pthread_t t_link;
  .....
  if(is_root == 1) {
   pthread_create(&t_link, NULL, &rc_link_training, &_Gen3Pcie[inst_num]);
  }
  else {
   pthread_create(&t_link, NULL, &ep_link_training, &_Gen3Pcie[inst_num]);
  }
  pthread_join(t_link, NULL);
  .....
  return (1);
}
```

The C-method is finally imported and synchronized in the configure_phase of the UVM to coordinate things across the C and SV (UVM) domain.

```
import "DPI-C" context task XactorConfigure(int unsigned is_root, int unsigned inst_num);
```

```
 task configure_phase(uvm_phase phase);
   phase.raise_objection(this);
   super.configure_phase(phase);
   fork
     XactorConfigure(cfg.device_is_root, inst_num);
   join
   phase.drop_objection(this);
endtask
```

Profiling and Debugging DPI based communication in co-simulation context

An incorrect usage of formal/actual arguments can lead to memory leaks in the interface which can be quite difficult to debug. Hence, an appropriate profiling interface is required as well. Profiling schemes for assessing the optimal size of transfers, mechanisms for software based caching is required in the context of DPI based co-emulation but is not easy to address.

Various mechanisms which can be employed for profiling:

- Built-in machinery in x86 CPUs that counts executed instructions. Tools like “VTune” can access those counters.
- CPI measure would show us the average time spent on executing an instruction in both simulations as well as emulation.
- Use call counters displayed in the DPI functions. The DPI method with the biggest numbers shall indicate that they can be rewritten for better efficiency.
- Use $time in SV code before and after the calls

V. Sample results based on DPI based improvements

Finally, we list down improvements observed in simulation performance and memory overhead for the following setups:

1. A UVM frontend leveraging SV DPI to communicate with a Synthesizable PCIe transactor
2. DPI based scoreboard and reference for a video decoder block

Changes Done	Percentage Improvement
Improvement using less number of DPI calls : 100 secs for 35 DPI calls got reduced to 30 secs for 20	3.33X
Using efficient file IO operations : usage of standard file streaming leads to a reduction from 90 secs to 15 secs	6X
Loading binary files against hex files	5X
Introducing threading : in file loading operation – time improved from 50 secs to 10 secs	5X
Usage of correct and compact and native C data types	1.5X
Offline processing on the C-side	2X

VI. Acknowledgement

We would like to thank members of our R&D team - Jan Koslacz; Cedric Alquier; Frederic Dumoulin for providing their expert insights in this area. A significant number of recommendations have been made based on their inputs.

VII. References

[1] “IEEE Std. 1364-2001 standard for the Verilog Hardware Description Language”, IEEE, Pascataway, New Jersey, 2001. ISBN 0-7381-2827-9.
[2] ZeBu™ ZEMI-3 Manual
[3] “IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language” (http://standards.ieee.org/getieee/1800/download/1800-2012.pdf)
[4] http://www.cs.cmu.edu/~gilpin/c++/performance.html
[5]http://stackoverflow.com/questions/1135964/simple-pthread-c
[6] https://computing.llnl.gov/tutorials/pthreads/