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

Hari Vinod Balisetty, Parag Goel, Amit Sharma
DVCON-2014
Introduction: C Integration Support

- DPI - SystemVerilog Standard
  - An interface between SystemVerilog and a foreign programming language: C or C++

- Simple interface to C models
  - Allows SystemVerilog to call a C function just like any other native SystemVerilog function/task
  - Variables passed directly to/from C/C++
  - NO need to write PLI-like applications/wrappers

- Why DPI
  - Easy of use
  - Allows SystemVerilog to call a C function just like any other native SystemVerilog function/task
  - Direct interface provides better performance

- Support both functions and tasks
DPI: Two-way Communication

- **Import “DPI”**
  - SystemVerilog calling C/C++ functions

```SystemVerilog
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) {
  ...
}
```

- **Export “DPI”**
  - C calls SystemVerilog functions
  - C calls SystemVerilog (blocking) tasks

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

DPI: Pure vs. Context Declarations

- Pure
  - Only non-void functions with no output or inout arguments can be specified as pure
  - Result depend solely on the values of its input arguments
  - Prone to better optimization

- Context
  - Best-suited for co-simulation. Support for import-export nesting
  - Not a barrier for simulator optimizations

- By default import tasks and functions are non-context
  - Not a barrier for simulator optimizations
Co-simulation Using DPI: Transactor Based Verification

**Result:** The speed improvement over cycle-based can be orders of magnitude faster reaching tens of MHz.

**Step 1:** An ‘import’ task/function which carries the relevant data to the hardware side.

**Step II:** The Intermediate C layer maps HW/SW calls with the correct data types and

**Step III:** forwards the data and invokes the hardware time consuming method through an export task.
Achieving co-simulation performance

Testbench and DUT are synchronized on a cycle basis, and communication overhead occurs at each and every cycle.

With Transaction-Based Emulation, synchronization is done only when required.

**Challenge:** Reduce the inefficiencies in *Communication*

- Reduced Frequency of DPI calls
- Enabling efficient communication
  - Appropriate usage of language constructs
  - Recommendations for more efficient data transfers
- Improved communication schemes
  - Offline processing
  - Leveraging Multi-threading
  - Enabling concurrency in the ‘C’ domain
  - Efficient File I/O
Reduced Communication Overhead

Appropriate Usage Of Language Constructs

- Declare imported functions as ‘pure’ whenever possible, to allow for more optimizations.
- Preferably use data types that map to native C-data types.
- Avoid usage of scalar data types (bit/reg/logic) mapped to scalar “unsigned char”, and the packed counterparts mapped to canonical form.
- The SystemVerilog-specific types, including packed types (arrays, structures, unions), which have no natural correspondence in C.

<table>
<thead>
<tr>
<th>SystemVerilog</th>
<th>C (input)</th>
<th>C (out/inout)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>char</td>
<td>char*</td>
</tr>
<tr>
<td>shortint</td>
<td>short int</td>
<td>short int*</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>int*</td>
</tr>
<tr>
<td>longint</td>
<td>long int</td>
<td>long int*</td>
</tr>
<tr>
<td>shortreal</td>
<td>float</td>
<td>float*</td>
</tr>
<tr>
<td>real</td>
<td>double</td>
<td>double*</td>
</tr>
<tr>
<td>string</td>
<td>const char*</td>
<td>char**</td>
</tr>
<tr>
<td>string[n]</td>
<td>const char**</td>
<td>char**</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SystemVerilog</th>
<th>C (input)</th>
<th>C (out/inout)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>svBit</td>
<td>svBit*</td>
</tr>
<tr>
<td>logic, reg</td>
<td>svLogic</td>
<td>svLogic*</td>
</tr>
<tr>
<td>bit[N:0]</td>
<td>const svBitVecVal*</td>
<td>svBitVecVal*</td>
</tr>
<tr>
<td>reg[N:0] logic[N:0]</td>
<td>const svLogicVecVal*</td>
<td>svLogicVecVal*</td>
</tr>
<tr>
<td>Open array[ ]</td>
<td>const svOpenArrayHandle</td>
<td>svOpenArrayHandle</td>
</tr>
<tr>
<td>(import only)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chandle</td>
<td>const void*</td>
<td>void*</td>
</tr>
</tbody>
</table>
Reduced Communication Overhead

**Reduced Number of Argument in DPI calls**

- Each argument should be declared as narrow as possible
  - for example, never use the 'int' data type if the argument can only be 0/1.
- Combine them and prune argument lists to reduce the amount of data transferred from the simulator

```plaintext
class compare_values {
   context function void compare_values {
      (int unsigned mem_idx, int unsigned CAL_VAL, int unsigned ADDRESS);
   }
}
```

```plaintext
class compare_values {
   context function void compare_values {
      (bit[7:0] mem_idx, bit[15:0] CAL_VAL, bit[31:0] ADDRESS);
   }
}
```

```plaintext
define VA8_T byte unsigned // bit[7:0]
define VAL_T shortint unsigned // bit[15:0]
define ADD_T int unsigned // bit[31:0]
```
Reduced Communication Overhead

Reduced Frequency of DPI calls

• If DPI calls are very frequent,
  o Each DPI call results in a transfer of at least a n-bit packet header. Additional n-bit words are added to the packet to convey argument values.
  o When passing a limited number of constants, try to create multiple versions of DPI functions to eliminate the need for constants.

```c
import "DPI-C" context function void compare_values
(`VA8_T mem_idx, `VAL_T CAL_VAL, `ADD_T ADDRESS);
```

```c
import "DPI-C" function void compare_values_EDGE
(`ADD_T add, `VAL_T val);
```
Reduced Communication Overhead

Reduced Frequency of DPI calls

- Reduction of redundant calls, 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 \( 4 \times 2^{(n\text{-bit})} \) words. One way to minimize the total amount of words transferred is to combine calls. In the above example, by creating a new wrapper function \( \text{ffgg(bit a, bit b, bit c, bit d)} \), we can reduce the total transfer size from \( 4 \times 2^{(n\text{-bit})} \) words to \( 2^{(n\text{-bit})} \).

```plaintext
import "DPI-C" context function void compare_values
(`VA8_T mem_idx, `VAL_T CAL_VAL, `ADD_T ADDRESS);
```

Comparison of content for two memories
@ Same event and same address with different values

```plaintext
import "DPI-C" context function void compare_values
(`VA8_T mem_idx, `VAL_T CAL_VAL, `ADD_T ADDRESS);
```

Specialized function
- \( \rightarrow \) Infer mem_idx implicitly
- \( \rightarrow \) Reduced arguments
- \( \rightarrow \) Reduced number of call

```plaintext
import "DPI-C" function void compare_values_HV
(`ADD_T add, `VAL_T val_0, `VAL_T val_1);
```
Reduced Communication Overhead
Tips for efficient data transfers

To enhance data transfers to the host PC (System Verilog),
- Avoid unused bits by using the narrowest data type possible
- Avoid logic vector (4-state) arguments and unpacked structures
- Use arguments whose width is less than or equal to 32 bits
- Concatenate large numbers of short vectors in Verilog and unpack them on the C++ side. This will ensure more data transfer with less communication overhead.
- Bit-vectors aligned with 32-bit or multiple of 32-bits boundaries yields greater performance. So does the use of 64 bit wide integers as they are natively mapped as longint on C-side.

On the C-front:
- If the memory/array is 16-bit/8-bit wide then use appropriate data type for building C-side arrays. For example, use byte to map the 8-bit wide vectors, shortint for 16-bit wide vectors etc.
  - Using wider elements than needed is a bad idea for two reasons:
    - RAM usage goes up, and
    - CPU cache may become clogged with junk.
Reduced Communication Overhead

Communication schemes

• Offline processing
  o 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.

• Leveraging Multi-threading

```c
void* rc_lnk_trg(void* arg) {
  Pcie foo = *((Pcie*)(arg));
  foo.rc->runBFM(PCIE::RunUntilTrainingDone);
  return (NULL);
}
void* ep_lnk_trg(void* arg) {
  Pcie foo = *((Pcie*)(arg));
  foo.ep->runBFM(PCIE::RunUntilTrainingDone);
  return (NULL);
}
```

```c
int XactorConfigure(int is_root, int inst_n) {
  pthread_t _link;
  if(is_root == 1) {
    pthread_create(&_link, NULL, &rc_lnk_trg, &Pcie[inst_n]);
  } else {
    pthread_create(&_link, NULL, &ep_lnk_trg, &Pcie[inst_n]);
  }
  pthread_join(_link, NULL);
  return (1);
}
```

```c
import "DPI-C" context task XactorConfigure(int is_root, int inst_n);
```
Reduced Communication Overhead

Communication schemes

- **Efficient File Operations**
  - **Original Case:** In this example we have 56 variable sized memories that needs to be loaded using the HEX dump files.

```c
int* PAT_MEM_AA; // [(H_PIXEL )];
int* PAT_MEM_AB; // [(H_PIXEL )];
...
int* PAT_MEM_BA; // [(T_PIXEL*3)];
int* PAT_MEM_BB; // [(T_PIXEL*3)];
...
int* PAT_MEM_CA; // [(T_PIXEL )];
int* PAT_MEM_CB; // [(T_PIXEL )];
...
```

**Loading Time**

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

Using efficient File IO
Results & Summary

Improvements observed in simulation performance and memory overhead for the following setups:

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

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