UVM Testbench Considerations for Acceleration

Kathleen A Meade *(Author)*
Solutions Architect
Cadence Design Systems, Inc
San Jose, CA
meade@cadence.com

***Abstract*— UVM has quickly become a universal standard for developing efficient, exhaustive verification environments for block-level and system-level designs. One of the key principles of UVM is to develop and leverage reusable verification components. One area of reuse that is gaining momentum is the reuse of verification components and environments between standard simulators and hardware acceleration.**

This paper focuses on techniques used to take an existing UVM testbench and make the required changes to run with hardware acceleration. If these considerations are made during testbench development, migration to hardware acceleration as a performance option is much easier. This paper offers recommendations for making your UVM environment (and components) reusable for acceleration.

Keywords—UVM; Hardware Acceleration; SystemVerilog; Performance

I. Introduction

The Universal Verification Methodology (UVM) has become widely adopted for development of complex verification environments and many teams have begun to reuse verification components and environments for second and third generation designs. When verification projects require very long simulation times, hardware acceleration can be leveraged to reduce simulation time from hours to minutes or even days to hours. Ideally, a user would like to be able to run with the same UVM testbench (TB) in both simulation and hardware acceleration modes ensuring there is only one development stream to manage.

Methodology adjustments discussed in the paper for the development of the UVM verification environment will not have a negative performance impact for pure simulation and can have a positive impact when the design is moved to acceleration hardware. Some of the performance considerations that are covered:

- Partitioning the top-level module so that the interfaces, DUT and additional synthesizable components are in one module and the testbench is in a separate module.
- Separating your monitor into a collector (for signal-level information) and a monitor (for checking/coverage).
- Limiting access between the DUT and the testbench. Limit access to the DUT to the driver and collector only.
- Designing the environment to minimize the number of transactions between the drivers/collectors and the rest of the environment. Group transactions that can be executed without interaction with the rest of the testbench.
- Creating synthesizable tasks to take transactions and drive signals (driver) and to reassemble transactions from signal-level details (collector). These tasks can be moved to your interface and called from the driver/collector. Interfaces should be synthesizable so they can be partitioned to the hardware side.
- Removing timing from sequences. Use alternate methods of specifying delays between and within sequences. A timing agent is introduced to execute “delay” sequences. These sequences execute in the interface during simulation and on the hardware side during acceleration.
- Optimizing randomization of transactions. Make sure that unnecessary calls to randomize are not being executed. These calls can have a significant impact on performance.

II. Is Hardware Acceleration a Good Performance Option for Your UVM Testbench?

Before considering acceleration, it is important to profile the environment with a reasonably long simulation runtime to check that a significant portion of the time is being spent in the DUT code. Generally speaking, acceleration is not an efficient performance option for environments where the testbench time is significant and the DUT time is small. However, when transitioning to sub-system and system-level verification, the verification focus may not require the detailed verification features that are otherwise requisite at the block/IP level. This means that, even in cases where the simulation profile shows that the testbench time is large, testbench optimization techniques can be applied and those environments can be candidates for acceleration also.

Acceleration is also not a good option if your simulation run times are short.

The main performance factors to consider are described here and shown in Figure 1 below:

1. <u>UVM Testbench Runtime</u>: This comprises any time spent in the simulator, including randomization of the testbench, configuration and building the verification environment, configuring the DUT, driving random stimulus, checking received data against a reference model and sampling coverage. For a long simulation, the testbench configuration and build time should be negligible since it only occurs once, at the beginning of simulation.

2. SW/HW Synchronizations: Signals and transactions that are sent between the UVM testbench and the DUT initiate a synchronization event. Every interaction between the simulation and hardware incurs a delay. Additionally, performance is impacted when data needs to be "moved" between the simulator and the hardware. This overhead can be critical for performance.

3. DUT Runtime: This includes any aspect of the design (and testbench) that is executed on the acceleration hardware. Acceleration hardware is very fast so it is important to move as much of the design (and testbench) to the hardware side as possible. The caveat is that anything on the HW side must be synthesizable.

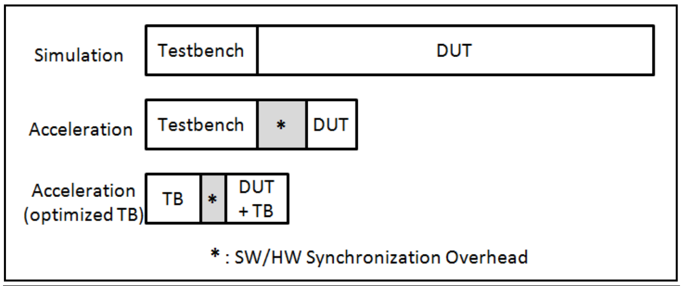

Fig. 1. Relative Runtimes for Simulation and Acceleration

For example, if you profile a simulation run and the testbench takes 25% of simulation time, the DUT is 60%, and the synchronization estimation is 15%, the maximum performance boost, if the DUT time is reduced to zero, is calculated as follows:

*HW_TIME = 60% + 60%*15% = 69%*

Estimated speedup (without TB optimization)
= (100/(100-HW_TIME)) = 3.2X

In this case, hardware acceleration may not be a good option.

If the DUT time is increased to 85% and the overhead is reduced to 10%, the maximum performance boost improves:

*HW_TIME = 85%+85%*10% = 93.5%*

Estimated speedup (without additional TB optimization)
= 100/6.5 = 15.4X

Once you establish that the design is a good candidate for acceleration, there are adjustments that can be made to improve performance beyond these numbers. Additional acceleration performance is achieved by reducing the time spent generating random stimulus and transferring data between the hardware-side and the software-side.

III. Partition Your Environment

It is recommended to partition your top-level module so that the DUT and additional synthesizable aspects of the environment are in one module and the UVM testbench is in a separate module. This should not impact simulation performance.

Figure 2 shows a simplified verification environment, partitioned to support hardware acceleration.

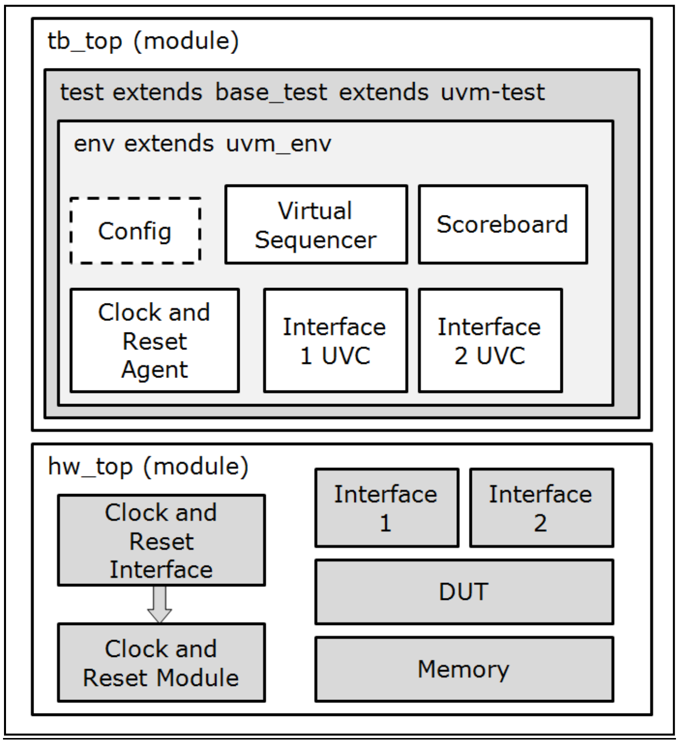

Fig. 2. Simplified verification environment with DUT and TB separated.

The top-level hardware module (hw_top) includes the DUT instance, a clock generator, interfaces that connect between the testbench and DUT and, optionally, synthesizable memory components. The testbench module (tb_top) is simplified to include the UVM package, UVC component packages, and an initial block to configure and start the test. The only purpose of the tb_top module is to use the UVM configuration database to configure the virtual interfaces and call the run_test() command.

```
module tb_top();
 import uvm_pkg::*;
 `include "uvm_macros.svh"

 import my_uvc_1_pkg::*;
 import my_uvc_2_pkg::*;
 import clk_reset_pkg::*;
 import my_tb_pkg::*;

 initial begin
  uvm_config_db#(virtual my_uvc_1_if)::set(
               null, "*.my_uvc_1*", "vif",
                hw_top.my_uvc_1_if0);
  uvm_config_db#(virtual my_uvc_2_if)::set(
               null, "*.my_uvc_2*", "vif",
               hw_top.my_uvc2_if0);
  uvm_config_db#(virtual clk_reset_if)::set(
               null, "*", "clk_reset_if",
               hw_top.clk_reset_if0);
  run_test();
 end
endmodule
```

Fig. 3. Code example for simplified tb_top module.

```
module hw_top();
 // Interface Instances
 my_uvc_1_if  my_uvc_1_if0  (...);
 my_ucv_2_if  my_uvc_2_if0  (...);
 clk_reset_if clk_reset_if0 (...);

 //Clock/reset generator, memory (opt)
 clkgen clkgen(clk_reset_if0);
 memory mem_inst (...);

 // DUT instance
 top_dut dut( .clk_rst_if(clk_reset_if0),
              .uvc_1_if(my_uvc_1_if0),
              .uvc_2_if(my_uvc_2_if0),
              ...);
endmodule
```

Fig. 4. Code example for hw_top module

IV. Implement a Collector and a Monitor

UVM recommends inclusion of a monitor in each agent that passively samples signals on the DUT interface, assembles this information into transactions, collects coverage and performs checking.

For hardware acceleration (and simulation), it is recommended to split the monitor activity into signal- and transaction-level activities. This is similar to the UVM recommendation for creation of stimuli. For generating stimuli, UVM enforces a separation between the transaction level (sequence/sequencer) and the signal-level activity (driver).

On the monitoring side, this separation is done by splitting the monitor into a low-level collector class and a high-level monitor that does transaction-level coverage and checking. The collector is also a passive entity. It follows the interface-specific protocol to capture transactions from signal data. The transactions are passed to the monitor by way of UVM TLM ports and the monitor can do checking, sample coverage, and pass the transaction off to the scoreboard and any other testbench component which may want access.

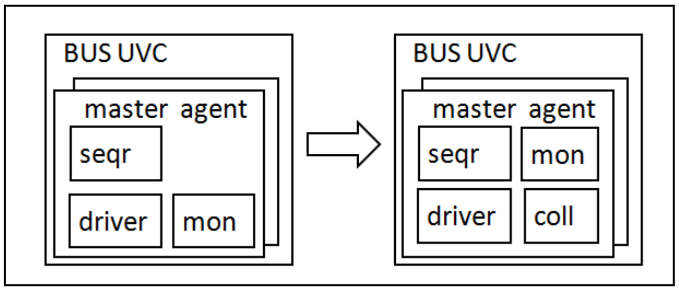

Fig. 5. Recommended UVC structure with collector and monitor

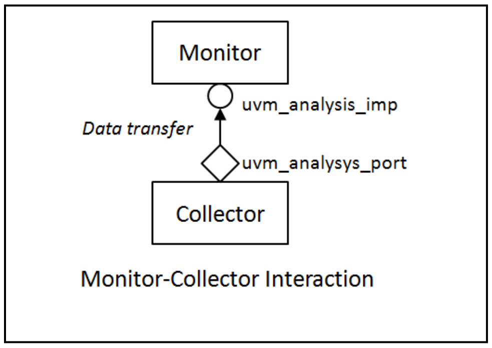

Fig. 6. Monitor/collector interaction

V. Minimize Interactions Between the Testbench and DUT

When hardware acceleration is deployed, every interaction between the DUT and the testbench requires a synchronization event, and this adversely affects performance. It is important to limit access between the testbench and DUT to the driver and the collector ONLY. This sounds intuitive, but it is difficult to enforce this policy. Every interaction outside the driver/collector must be addressed when the design and testbench are migrated to run on acceleration.

Two common places where this is found is waiting for signals to change inside a sequence, and waiting for reset. A simple UVM clock and reset agent and/or an interrupt agent can be added to the UVM testbench to limit access to these signals and generate events related to them.

VI. Driver and Monitor Adjustments

When users migrate from simulation to hardware acceleration, the first step is usually to partition the UVM testbench to the simulator, move the DUT to the hardware side and verify that the same tests run in both simulation and acceleration. This simple, signal-based solution takes less time to implement and can produce results in the 5X to 20X range, depending on the implementation details of the testbench and DUT.

With transaction-based acceleration, part of the testbench is also moved into the hardware accelerator and the interface between the testbench and the DUT is through task calls rather than individual signal transitions. This further reduces the number of synchronizations between the simulator and accelerator and can produce results in the 30-300X range. This performance boost is desirable, but requires that any portion of the testbench that goes to the hardware side is synthesizable.

Development of your interface UVC driver usually includes implementation of a “drive_transfer” task which converts a transaction to a series of signal interactions over a period of time. These signals are connected to the DUT through a virtual interface. Similarly, the UVC monitor/collector typically includes a “collect_transfer” task which captures signal-level details from the virtual interface and re-assembles transactions for checking, coverage and analysis.

Figure 7 illustrates the structure of a traditional UVC. Signals are driven and clocks are referenced from the driver and collector class through a virtual interface handle.

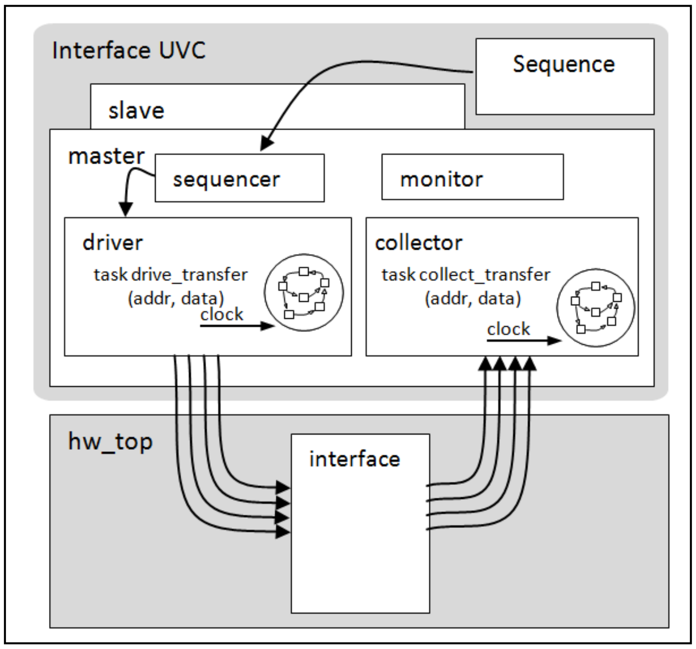

Fig. 7. Traditional UVC Structure

For hardware acceleration, it is recommended to develop the "drive_transfer" and "collect_transfer" tasks to be synthesizable so they can be moved to the DUT interface and included in the hw_top module of the environment. Optionally, these methods can be placed in the interface for both simulation and acceleration. The only drawback to having these tasks inside the interface instead of the driver class is that you cannot use the UVM factory to override the default behavior of those tasks. This requirement (an override of the drive_transfer and/or collect_transfer tasks) is rarely seen in UVM environments.

Figure 8 illustrates the structure of an acceleratable UVC. The driver and collector call time consuming interface tasks through a virtual interface handle. Signals are driven and clocks (and signals) are referenced directly inside the interface.

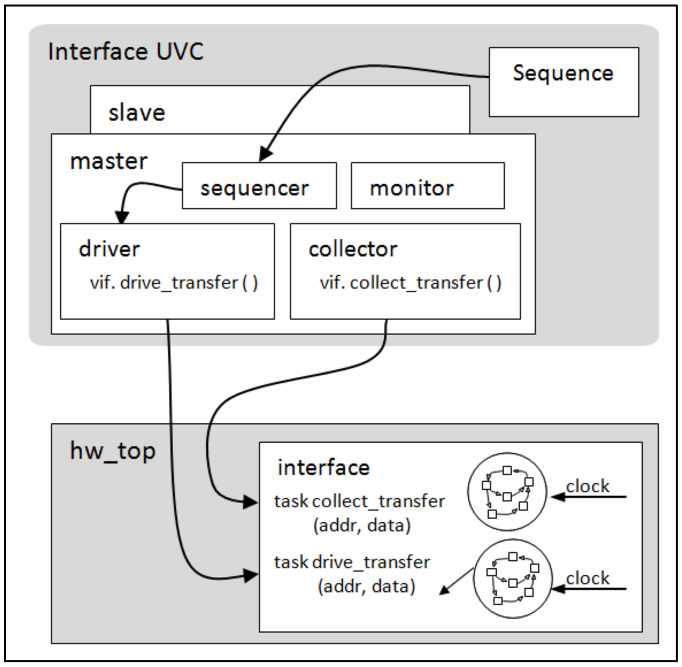

Fig. 8. Accelerated UVC Structure

VII. Clocks and Reset

Clocks and resets are an important part of designing for acceleration because the clock signals toggle frequently and can cause undesired synchronization events. It is recommended to implement a simple reusable UVM agent which uses sequences to configure and start clocks, initiate resets and execute delay sequences from the UVM testbench. The implementation details of generating clocks, delays and resets will be handled in the SystemVerilog interface and a clock generation module on the hardware side. The UVM clock and reset agent architecture is illustrated in Figure 9.

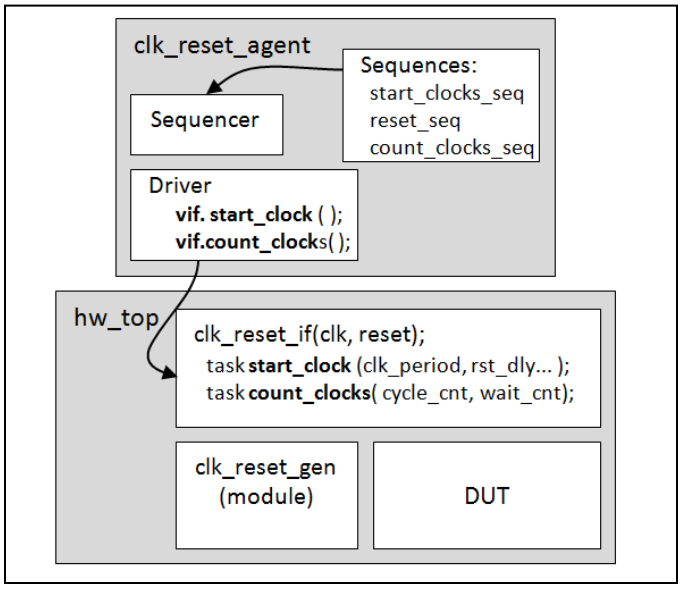

Fig. 9. UVM Reusable Clock and Reset Agent

The UVM code below demonstrates a simple reusable data item that can be used to control the clock and reset generation in the DUT. It has control knobs indicating clock period, reset information and a cycle count for issuing a delay or timeout.

```
class clk_reset_item extends
                      uvm_sequence_item;
int clk_period;
int reset_delay;
int cycle_count;
bit wait_count;
bit run_clk;

// UVM utility macros
`uvm_object_utils_begin(clk_reset_item)
  `uvm_field_int(clk_period, UVM_DEFAULT)
  `uvm_field_int(reset_delay, UVM_DEFAULT)
  `uvm_field_int(cycle_count, UVM_DEFAULT)
  `uvm_field_int(wait_count, UVM_DEFAULT)
  `uvm_field_int(run_clk, UVM_DEFAULT)
`uvm_object_utils_end

  // constructor
function new (string name="");
  super.new(name);
endfunction: new

endclass: clk_reset_item
```

Fig. 10. UVM code for clock and reset data item.

In the driver's connect_phase(), it gets the clock and reset interface from the configuration database. Then in the run_phase() it gets clk_reset_items and uses the control information to execute clocking and reset tasks implemented in the interface.

The SystemVerilog code in Figure 11 below demonstrates the techniques used in the driver to use the clk_reset_item information to call the appropriate interface tasks.

```
class clk_reset_driver extends
                 uvm_driver#(clk_reset_item);
...
virtual clk_reset_if vif;
...
function void connect_phase(uvm_phase phase);
 super.connect_phase(phase);
 if (!uvm_config_db#(virtual clk_reset_if)::
       get(this, "*", "clk_reset_if", vif))
 `uvm_error("NO_VIF",
             "Virtual interface not set!")
 endfunction : connect_phase

task run_phase(uvm_phase phase);
...
 forever begin
  seq_item_port.get_next_item(req);
  if ((req.run_clock ==1)||(reset_delay !=0))
    vif.start_clock(req.clk_period,
            req.reset_delay, req.run_clock);
  else vif.count_clocks(req.cycle_count,
                      req.wait_count);
  ...
  seq_item_port.item_done();
end
...
endtask : run_phase

endclass : clk_reset_driver
```

Fig. 11. Code for the clock and reset driver connect_phase and run_phase

To ensure that the clocks are correctly recognized, the clock generating structure is placed inside a module. The control of the clocks generated by that structure is handled from an interface that passes the appropriate control signals to the module.

Handling timeouts is a particular concern in accelerated environments. It is often necessary to check that a particular event, transaction or interrupt occurs within a certain time. If clocks are referenced on the class based side this will require frequent (expensive) synchronizations between the accelerator and the simulator. To avoid this, a timer and sequence to drive it is built into the clock and reset UVC. The sequence returns once the set timeout has been reached and an event has been emitted. There is a field that can be set to cause an error if the timeout is not reached before the end of the test. The timer can handle concurrent timeout requests, even if they finish simultaneously.

The SystemVerilog code shown in these examples was simplified to demonstrate the techniques. If it is a design with reuse in mind, this clock and reset UVM agent can be used within and across many projects and many teams.

VIII. REMOVING TIMING FROM SEQUENCES

For optimum performance it is important to make sure that there is no timing in the testbech. This includes references to clocks, #delays and waiting on transition of a DUT signal. Use alternate methods of specifying delays between and within sequences. The timing agent can execute "delay" sequences.

These sequences execute in the interface during simulation and on the hardware side during acceleration.

The clock and reset agent timer that counts clocks for timeouts can also wait for the number of clocks to execute before it returns while blocking the testbench from continuing execution.

The clock and reset agent also includes a sequence that can be configured to wait (or not wait) for a user defined delay of a variable number of clocks. This sequence can be used within a virtual sequence, as shown in the SystemVerilog code below.

```
class traffic_vseq extends uvm_sequence;

 //`uvm_object_utils and constructor

 // UVC sequences
 clock_and_reset_start_seq clock_seq;
 clock_and_reset_delay_seq delay_seq;

 my_uvc1_config_seq  uvc1_config_seq;
 my_uvc2_traffic_seq  uvc2_traffic_seq;

 my_memory_load_seq  load_mem_seq;

 task body();
  `uvm_create(clock_seq)
   // Start Clocks and Reset
   clock_seq.clk_period = 10;
   clock_seq.reset_delay = 10;
   clock_seq.run_clock = 1;
   `uvm_send_on(clock_seq,
                 p_sequencer.clk_reset_seqr)
   // Load Memory
   `uvm_do_on(load_mem_seq,
               p_sequencer.mem_seqr)
   // Configure
   `uvm_do_on(uvc1_config_seq,
               p_sequencer.uvc1_master_seqr)
   // Run Traffic
   `uvm_do_on(uvc2_traffic_seq,
               p_sequencer.uvc2_master_seqr)
   // Wait 30 clocks to end simulation
   `uvm_create(delay_seq)
   delay_seq.clock_cycles = 30;
   delay_seq.wait_count = 1;
   `uvm_send_on(delay_seq,
                 p_sequencer.clk_reset_seqr)
  endtask
endclass : traffic_vseq
```

Fig. 13. Virtual sequence executing timing sequence

The code in Figure 12 replaces the following code that might found in a virtual sequence:

```
task body();
  @(negedge p_sequencer.vif.reset)
  // Load Memory, configure, run traffic
  // Wait 30 clocks to end simulation
  repeat(30)
    @(posedge p_sequencer.vif.clock);
endtask
```

Fig. 12. Virtual sequence body example with timing.

IX. Randomization Adjustments

Make sure that the random constraints are as simple as possible. If randomization is not required when you are running tests with acceleration, make sure your interface UVC includes a sequence that does not call randomize().

Summary

Spending time up-front to construct UVM environments that are easily portable to hardware acceleration can reap big performance benefits in the end. The recommendations we make are enhancements to the UVM that meet the goal of using a single environment for simulation and hardware acceleration.

Acknowledgement

I would like to thank my colleagues at Cadence Design Systems, Inc for input and suggestions for this paper.

References

[1] D. Cohen and P. Edstrom, "Developing a single UVM environment for software simulation and hardware acceleration," http://support.cadence.com (Rapid Adoption Kits).

[2] K. Meade and S. Rosenberg, "A Practical Guide to Adopting the Universal Verification Methodology (UVM) Second Edition ", United States of America, 2013.

[3] W Queen, J Sprague and J Pierce, Unconstrained UVM SystemVerilog Performance, DVCon Proceedings, 2012