Accelerated simulation through design partition and HDL to C++ compilation

Theta Yang, Sga Sun
Advanced Micro Devices
1387 Zhangdong Road, Shanghai, China 201203

***Abstract*-System-level simulation for a complex SOC design usually suffers from bad simulation performance and long debug turn-around time. As a result only limited system-level simulation can be executed before the design tape-out. In this paper, we present an accelerated simulation methodology through design partition and HDL to C++ compilation. With this method, a large chip design is partitioned into smaller blocks, and with Verilator[1], an open source HDL compiler and simulator, each block is converted into its C++ representation and then compiled into shared object code. The shared object code can be then invoked by simulation to perform as the block's functional model. A C++ based top-level module is used to assemble all the generated block models, with method's such as cycle-based simulation and model input change detection, the communication between these block models in the simulation can be largely suppressed and simulation performance considerably boosted. The resultant system model is an ideal design model that can be used in system-level simulation.**

I. Introduction

Design simulation at RTL level for complex SOC is not suitable to exercise system-level scenarios such as OS booting, as a result, hardware acceleration or FPGA emulation have to be planned to provide enough system-level verification coverage before the design is taped out. However those solutions are relatively expensive compared with traditional simulation and hard to setup. There are several published applications which use Verilator [4] to do fast simulation or emulations. Several are listed in the references. In [5], a high performance cycle accurate SystemC model has been generated by Verilator for an embedded processor. In [6] a parallel simulation based on simulation server's multi core and Verilator has been proposed which could provide simulation speed up. In our paper, we propose a method to generate a fast system-level design model from the original HDL design, the method has several steps including, design partition, block-level HDL to C++ compilation, and the top-level assembly. The generated design model is cycle accurate in behavior but in addition also has a good simulation performance to exercise system-level scenarios.

A. Design partition

Most of the design synthesis tools can flatten and regroup design module hierarchies after reading in and elaborating the design. There are HDL compilers, such as Verific Verilog compiler [2], which also have such a capability. The compiler can read in a design specified in Verilog or VHDL, and generate a design with changed module hierarchies but equivalent in logical behavior. Similar technology has been used here but we use an in house tool to manipulate the design partition. After this step, the whole design is partitioned into a list of block-level designs, with similar complexity or gate count. These blocks can be manipulated by the following steps. The design partition step has two proposes. First, modern SOC design usually follows a component-based design and verification methodology. Third party IP's and in-house developed IP's are used as building blocks to be integrated at SOC level. By building a simulation model for each IP, and then assemble each IPs' the simulation model at SOC level. Each time after any design change, only the changed IP need to be re-compiled. Secondary, the input/output boundary signals of a design block need to be preserved from compiler optimization, so that the test bench can control or observe the signals.

B. HDL to C++ Compilation

An open source Verilog compiler and simulator called Verilator is used here to convert the HDL description of the design block to C++. The tool is designed for projects where fast simulation performance is of primary concern and which can also support the synthesis subset of Verilog. Rather than simply translation, Verilator can compile the code into much faster optimized model. The generated block-level models have a common invoke function which can be called by the top-level to evaluate the internal and output state of the block if the input signals' values have been set.

C. Top-level assemble

Two different top-level modules can be supported. In simulation mode, each generated block-level model is wrapped with a Verilog stub module, the stub module has same input and output ports with the original block top model, but with no internal logic. The invoke function of C++ model is imported as a DPI function into the HDL simulator. The stub module and the C++ model performs the behavior of the block and a HDL simulator is used to run the simulation. The benefit of running the generated system model in this simulation mode is that we can still simulate with the ordinary UVM [3] testbench code or design components with behavioral model in Verilog such as PLL or Serdes. Since these models are not coded in synthesis subset of HDL code, Verilator cannot compile them. The original design top-level is used to connect all sub blocks, and used as the top-level module by the simulation.

In emulation mode, each generated block model is wrapped with a C++ wrapper. The communication between block-level models can be largely suppressed if only the clock and asynchronous signal are treated as the evaluation trigger input of the block. The communications of other synchronous connection are through a global data structure which is signal value database shared between all blocks. The concept of 'cycle-based simulation' is followed here. In each cycle, a common set of callbacks are called serially for each block model. The callbacks are implemented in model method, and a simulations cycle comprised of serial phases including monitor phase, evaluation phase and drive phase. In monitor phase, the value of input signals will be obtained from the global signal data structure; in evaluation phase, the evaluation function that each block-level model has is called; and in drive phase, the output value of the output port for each block model is written back to the global signal data structure. To facilitate the 'cycle-based simulation' concept, a common shared minimum clock tick has been defined, with a frequency equal to the least common multiple value of all clocks in the design. In each tick, each block may evaluate once. The evaluation of the functionality of each block could be masked for the input event clock, if in the cycle there is no input signal changes for the block. This method are called ‘block input change detection’ and has been implemented in our C++ based top-level module. By these optimizations, the simulation performance is speeded up effectively.

II. ACCELERATED SIMULATION DESIGN MODEL BUILD

A. Verilator Description

Verilator is an open source software tool which converts Verilog to a cycle-accurate behavioral model in C++ or SystemC. It is restricted to modeling the synthesizable subset of Verilog and the generated models are cycle-accurate and 2-state. As a consequence the models typically offers higher performance than the widely used event driven simulators, however later can process the entire Verilog language and model behavior within the clock cycle. Verilator is now used within academic research, open source projects or for commercial semiconductor development. The below diagram shows the internal structure of the Verilator generated model, and the function *eval()* can be invoked to evaluate its behavior.

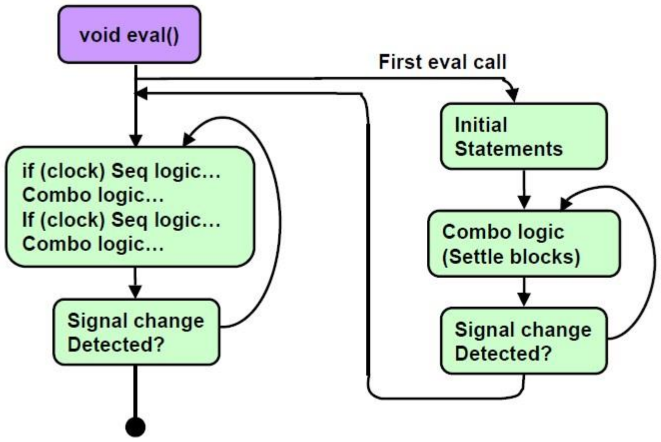
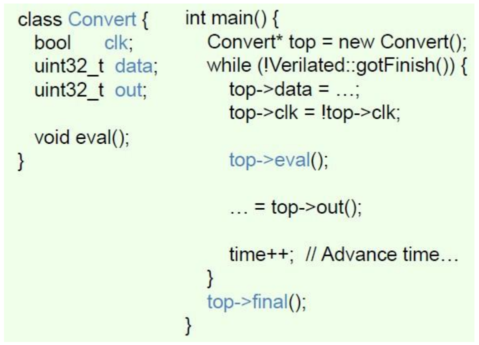


Fig 1. Diagram of Verilator generated block model

After a design model is compiled into C++, user application can create the generated model class and call the evaluation function in a loop. Verilator ignores the delay specified in the design which is the key difference from most of the hardware simulators. The below program sample is to demonstrate how a Verilated design model is used.



```
class Convert {          int main() {
   bool      clk;           Convert* top = new Convert();
   uint32_t  data;          while (!Verilated::gotFinish()) {
   uint32_t  out;              top->data = …;
                               top->clk = !top->clk;
   void eval();
}                              top->eval();

                               … = top->out();

                               time++;  // Advance time…
                            }
                            top->final();
                         }
```

Table 1. Verilated model usage example

B. Design Partition Flow

We use an in house tool to partition a big design into multiple sub-designs. The user of the tool creates a connectivity specification of design blocks and the tool will assemble these blocks and generate the corresponding HDL description from the specifications provided. This design connectivity tool supports making connections between blocks using abstracted interfaces, or sets of signals. A block can make connection to the outside in the form of ports. A block can be a leaf node, if it is at the bottom of the hierarchy and contains no sub-blocks. And a block can also be a containers which contains instances of other block.

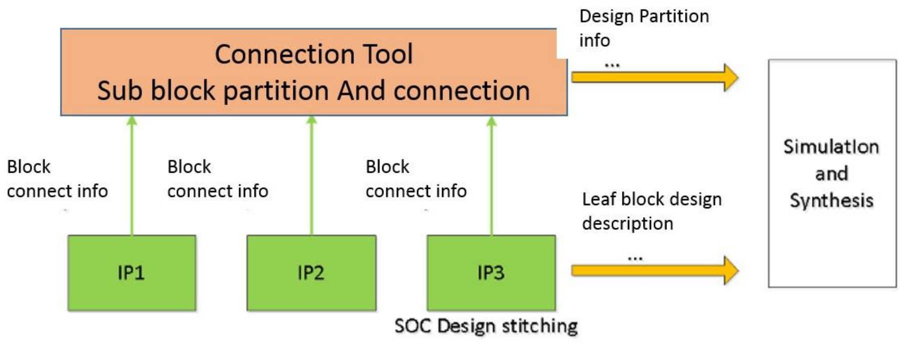


Fig 2. Diagram of design partition

The tool supports the changes of the design hierarchy by specifying the design partition information to it. As per the information provided the tool flattens the container blocks or groups multiple blocks to create a new container block. The output of the tool is a design with two levels of hierarchy, an automatically generated top-level in HDL format, and a set of leaf designs with each composed from original available HDL design. The tool also generates file lists for each block for simulation or synthesis compilation.

C. Model Assembly for Simulation

For each block design, the verilator compiler compiles it from a Verilog description to a C++ description. The block design in C++ description is then linked into shared objects, which can be loaded into a simulation or emulation system. To make the block design model capable enough to support both simulation and emulation, each block is wrapped inside a wrapper class which is compiled together into the shared objects. The wrapper provides common API for each sub-design, the major API includes, *monitor_input*, *evaluate* and *drive_output*. The below block diagram shows how each sub-design can be used in a larger system.

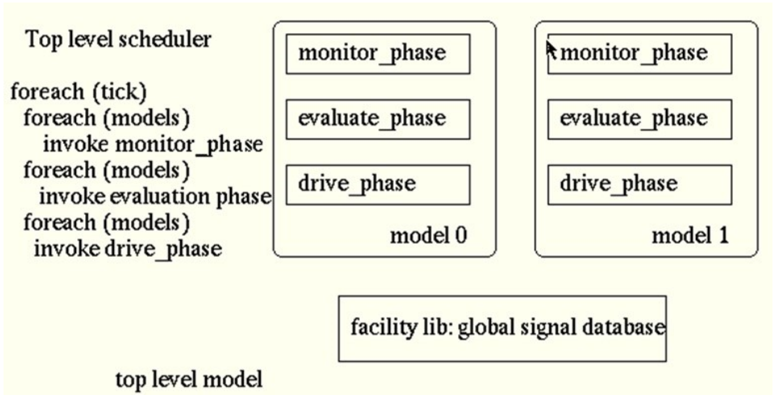
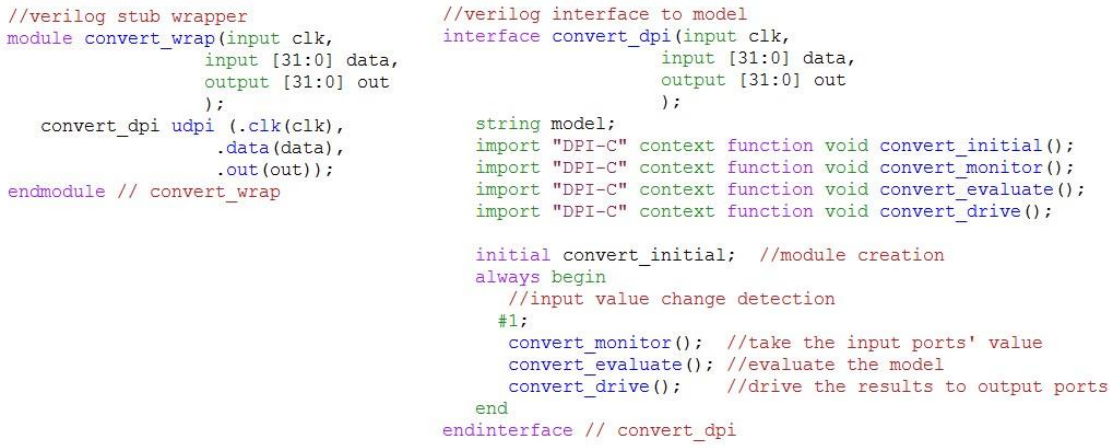


Fig 3. Diagram of top-level model

The Verilog and C++ co-simulation is achieved through system Verilog DPI [4]. For each C++ model, a Verilog stub is developed to provide the external connection of the model. There is an initial process inside Verilog stub which calls the related DPI to create the C++ model. And also inside the stub, there are processes to monitor the value change of all of the input ports. In case there are any input changes, the new value is achieved and set into the C++ model through DPI call, the C++ model will be evaluated by it and the output of the model will be driven to Verilog side. The model evaluation needs to be called every simulation delta cycle to sync with the Verilog interface value changes. This may slow down the simulation. We add extra input value change detection logic to avoid the model evaluation be called each delta cycle. In case there is no value change compared with last simulation delta cycle, the model evaluation is skipped.



```
//verilog stub wrapper
module convert_wrap(input clk,
                    input [31:0] data,
                    output [31:0] out
                    );
   convert_dpi udpi (.clk(clk),
                     .data(data),
                     .out(out));
endmodule // convert_wrap
```

```
//verilog interface to model
interface convert_dpi(input clk,
                      input [31:0] data,
                      output [31:0] out
                      );
   string model;
   import "DPI-C" context function void convert_initial();
   import "DPI-C" context function void convert_monitor();
   import "DPI-C" context function void convert_evaluate();
   import "DPI-C" context function void convert_drive();

   initial convert_initial;  //module creation
   always begin
      //input value change detection
     #1;
      convert_monitor();  //take the input ports' value
      convert_evaluate(); //evaluate the model
      convert_drive();    //drive the results to output ports
   end
endinterface // convert_dpi
```

Table 2. Verilated model wrapper for verilog simulation

In simulation mode, the whole design is composed by C++ sub-design models, the Verilog stub of those design models, and top-level design in Verilog. A Verilog simulator is also required to simulate the whole design. However, since all the model interface signals are preserved, the design hierarchy doesn't change, and the original test bench framework can still work. In addition, if the test bench is also in System Verilog or UVM based, then System Verilog is definitely required.

D. Model Assembly for Acceleration

The C++ based top-level model is used in acceleration mode. Since all of the design model is in C++, the HDL simulator is not required and so we implement the cycle-based simulation concept. Cycle-based simulation has very high performance for the simulation of synchronous design, but it is confined to synchronous design and it is not as timing accurate as event-driven algorithm. Also the cycle simulation scheduler is relatively easy to build compared with the event based simulation scheduler. Our target design is synchronous with very few asynchronous signal. In the cycle-based simulation, these asynchronous signals can only change the value with the minimum definition of cycle, it doesn’t cause any functionality failure in the simulation but it cannot simulate some physical device scenarios such as meta-stability of a register. Since we use our emulation system to simulate only a few system-level scenario instead of replacing the HDL functional simulation totally, the impact due to the limitation of our emulation system is mitigated.

Once all of the design models have been constructed, the simulation is executed under the control of the top-level simulation scheduler. The scheduler maintains all of the clocks in the design, and these clocks could have multiple phases. A clock provides a collection of stages where the block model can register callback functions. The collection of stages are defined and arranged for the specific purpose of performing simulation. For each clock phase, or "simulation tick", the scheduler proceeds loops through the collection of stages. Every model in the simulation will be visited. If a model is not gated, it will run through a clock tick and the registered callback function will be invoked.

From within their constructor, a block model object registries callbacks with particular stages. This means that whenever the specified stage is executed, the specified method is called on the block model. The implementations rely on a callback to be performed for every simulation tick. The callbacks are gated on the clock, and depending on the circumstances, it may be gated on the rising or the falling edge of the clock. This ensures that no matter what the clock frequency and phase, the callbacks will be triggered at the right moment. The flow diagrams below shows all of the simulation stage. A simulation will go through setup, initialization, execution, and completion stage. In the setup stage, the simulation models are instantiated and the global simulation events including clock and reset have been defined. In the initialization stage, the initialization value of the input signals or preset memory contents are initialized. The execution stage will run multiple clock ticks until the maximum required simulation cycles is reached or until some exit conditions occurs.

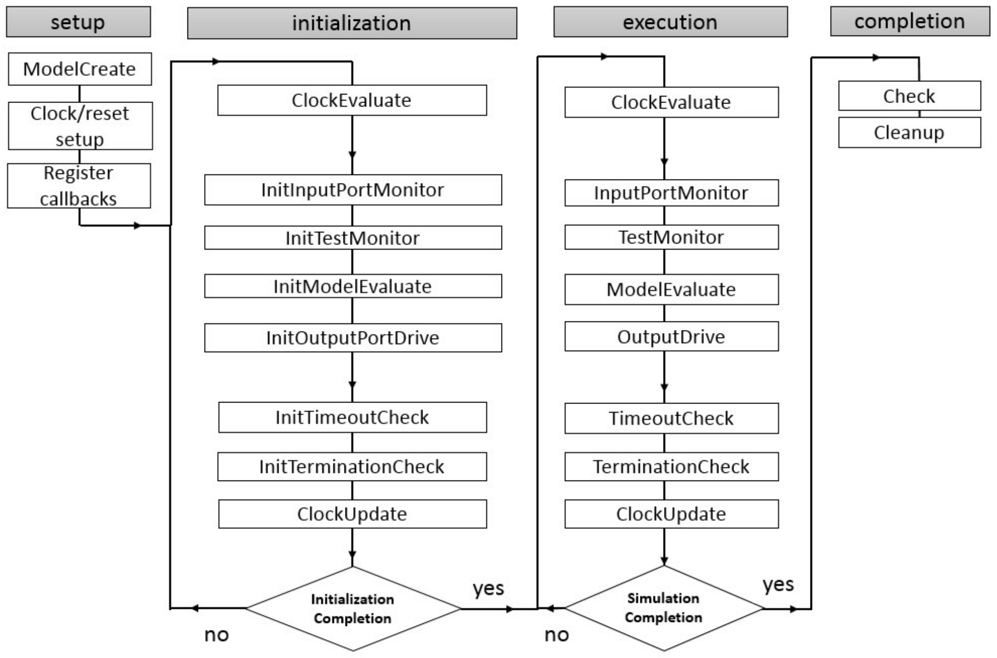


Fig 4. Flow diagram of the simulation scheduler

Clocks are modeled specially, and there are several classes for modeling clock signals within the simulation model. These classes allow users to construct clocks to control when user events are evaluated. These clocks keep track of the current simulation cycle, and cause time to be advanced within the simulation model. The clocks may be synchronized to, or drive, clock signals in the simulation model. A hierarchy of clock classes has been designed which is derived from *SimBaseClock*. A clock may be active or not, and that clock may gate user callbacks. *SimBaseClock* has an *evaluate*() function that is registered, and the callback will execute before any user callback. This way allows a clocks to gate any user registered callbacks. *SimBaseClock* has few attributes, which include running and startup delay. If the clock is not running, the associated event will never go active. The startup delay holds the clock off from any processing until the specified number of cycles have expired.

A clock can be multiple phase, and each phase is measured as a simulation tick. *SimRiseClock* will be active as an event when the current phase = 0. *SimFallClock* will be active as an event only when the current phase = phase of falling edge (subject to the offset). Utilizing these clocks as gates for user events and callbacks is a convenient method for constructing simulation environments that are efficient, and flexible to changes in the clock ratios. If the number of simulation cycles per clock cycle changes due to a change in the hardware configuration, the associated clock object will automatically adjust and call the user code on the appropriate cycle.

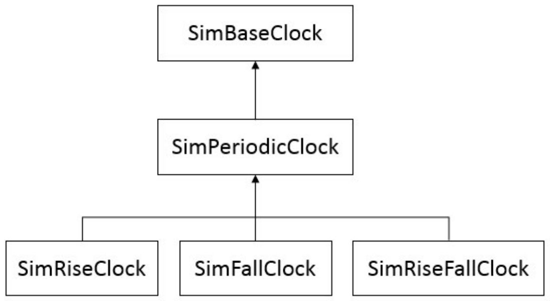
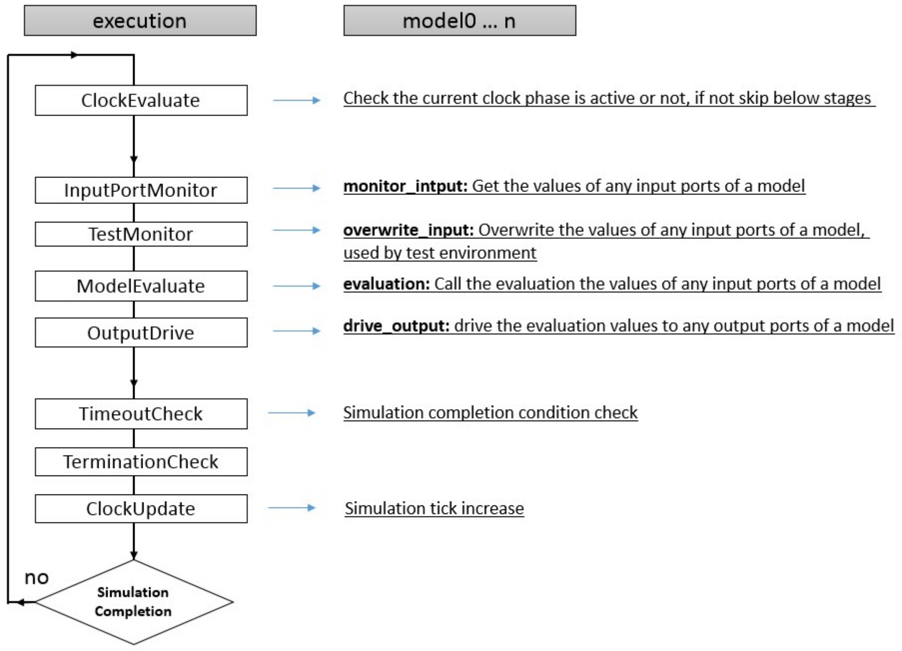


Fig 5. Diagram of the clock class hierarchy

Each sub-design model will register its API's into the proper clock events' callback. And for each simulation tick, the model clock event is evaluated first. If it is not gated, the model's callback function will be called. This way, the model will be evaluated for each clock it is associated with. There is a shared signal value data structure to store all of the top-level signals, for each tick in the monitor stage, the value is read into the design model. In the evaluation stage, if a model is not gated by its associated clock, the model will be evaluated. And finally in the driven stage, the output of each model is stored back into the central signal value structure.


execution
model0 … n
ClockEvaluate
Check the current clock phase is active or not, if not skip below stages
InputPortMonitor
monitor_intput: Get the values of any input ports of a model
TestMonitor
overwrite_input: Overwrite the values of any input ports of a model, used by test environment
ModelEvaluate
evaluation: Call the evaluation the values of any input ports of a model
OutputDrive
drive_output: drive the evaluation values to any output ports of a model
TimeoutCheck
Simulation completion condition check
TerminationCheck
ClockUpdate
Simulation tick increase
no
Simulation Completion
Fig 6. Callback stage definition of clock event and simulation

III. Performance Profiling

A. Prototyping System Configuration

The prototyping system for the simulation/emulation framework is a sub system of a complex SOC design, which contains on-chip interconnection fabric, controller IPs likes IO controller and on-chip peripherals. The block diagram of the system is as shown below in Figure 7. The original testbench is based on UVM methodology with the external interface being connected with different simulation models which with feed in the configuration and data path transactions. This entire system is then simulated under a Verilog HDL simulator.

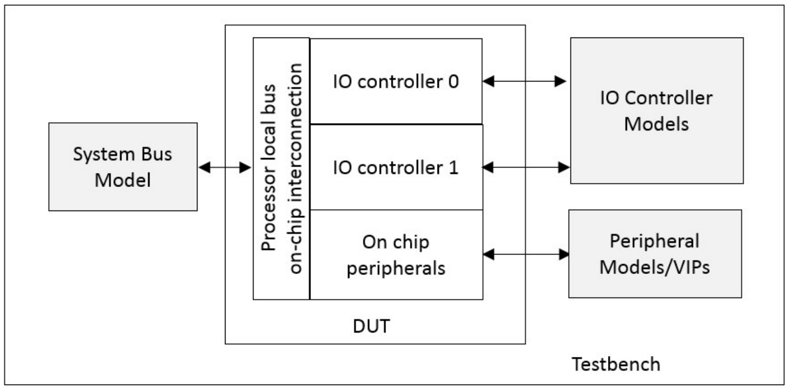


Fig 7. Prototype system

B. Simulation Setup

Both simulation mode and emulation mode of the Verilater generated IP models have been tested. In simulation mode, the original UVM based testbench can still be used with limited change. We can therefore take direct comparisons between simulations using the Verilated model or using original HDL description model. The below table shows the performance comparison between the HDL simulations and the Verilated model simulations on the same HDL simulator and server. Here we see the simulation performance is slightly increased with Verilated DUT model.

Test Case	Simulation with HDL design model (CPS)	Simulation with Verilated Model (CPS)
Sampe_0	8.8	10.4
Sample_1	8.4	11.0
Sample_2	7.9	9.2

Table 2. Performance for Verilated mode

C. Emulation Setup

In emulation mode the original testbench cannot be used and needs to be replaced with C++ BFM based testbench. The profiling information dumped from Verilog simulator contains the split-up percentage of simulation time consumed by the design model and the verification code. By adjust the profiling data, we can generate the cycles-per-second of performance for simulating the design model only on the commercial HDL simulator. The emulation mode test also generates the CPS for running the emulation model in the same server. The below table shows the performance comparison between HDL simulation and Verilated model emulation,

Test Case	Simulation with HDL design model (CPS)	Software Emulation with Verilated Model (CPS)
Sample_0	12.6	21.6
Sample_1	12.1	26.2
Sample_2	11.4	18.8

Table 3. Performance profiling result for emulation mode

In simulation mode, the simulator takes a large segment of the computation effort on testbench components besides the DUT. There is no significant improvement of simulation speed if we change the DUT from RTL model to Verilated model. And as the result of the CPS performance comparison shown in Table 2, The usage of Verilated simulation models can boost the simulation performance only slightly. In emulation mode, the simulation models in the original testbench have been replaced by simplified BFM models in C++, and even compared with scaled HDL simulation speeds, the resulting design model has is 2-3 times faster than simulation.

IV. CONCLUSION

A software-based accelerated simulation system has been proposed here which can be generated from HDL design with two key technologies, the HDL to C++ compilation and design partition. The generated block-level model can be assembled by two kinds of external wrappers. First, with the Verilog wrapper, the model can be used by ordinary simulation and secondly, with the C++ wrapper, the model can be used as a software-based emulation system. With the input change detection and cycle-based simulation technology, the simulation performance is optimized so as to be used in system-level simulation such as boot code simulation. The open source HDL simulator, Verilator, has been leveraged to perform block-level model generation. The example system is based on a subsystem-level design which includes a processor local bridge and IO controllers for a complex SoC. The subsystem design has been partitioned into several blocks, and the design model has been generated and demonstrated to work properly. A more exhaustive simulation system includes external CPU models which is under construction, and we believe this larger system-level model will run much faster than the original HDL simulation, and is promising to provide simulation performance adequate for system-level simulation.

Acknowledgment

The author would like to thank Mike Popovic from the management for encouraging us to work on technical initiatives. The author would also like to thank Vishwath Bhandary for cleaning up all grammatical issues in the draft paper.

References

[1] Wilson Snyder (2013) DVClub Bristol, Verilator: Open Simulation- Growing Up

[2] Verific Design Automation, http://www.verific.com

[3] Accellera, “Universal Verification Methodology,” available at http://www.accellera.org/downloads/standards/uvm

[4] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, IEEE Standard for System Verilog 18002012, New York, NY: IEEE, 2013

[5]Wilson Snyder, "Verilator: Fast, Free, But for me?" DVClub Presentation 2010, pp 11,

Available at: http://www.veripool.org/papers/Verilator_Fast_Free_Me_D VClub10_pres.pdf

[6] Embecosm Application Note 6. High Performance SoC Modeling with Verilator: A Tutorial for Cycle Accurate SystemC Model Creation and Optimization. Issue 1. Embecosm Limited, February 2009.

[7] Parallel Multi-core Verilog HDL Simulation based on Domain Partitioning, by ariq B. Ahmad , Maciej Ciesielski, 2014