A Methodology to Port a Complex Multi-Language Design and Testbench for Simulation Acceleration

Horace Chan
PMC
8555 Baxter Place,
Burnaby, BC, Canada,
V5A 4V7, 604-415-6000

Brian Vandegriend
PMC
8555 Baxter Place,
Burnaby, BC, Canada,
V5A 4V7, 604-415-6000

Efrat Shneydor
Cadence Design Systems, Inc.
2655 Seely Avenue
San Jose, CA 95134
408-943-1234

***Abstract*- PMC's verification teams started exploring simulation acceleration (SA) with hardware-assisted verification in 2011, as one of the early adopters of UVM Acceleration. They undertook this effort because of the complexity and size of their mixed-language designs, which were coded in SystemVerilog, Verilog, and VHDL, and stimulated using state-of-the-art testbenches coded in UVM-e.**

A few years later, the task of porting a design and testbench from simulation to acceleration evolved into a methodology and is now re-used across multiple verification teams. Finally, PMC has achieved the holy grail of SA, conquering the most complex challenges of SA verification including: 1) Speed – achieving 67x speed up, 2) Time to First Test – taking only a month to port a verification environment to run in acceleration mode, 3) Consistent – Running the same tests with RTL and an accelerated DUT, producing the same results.

This methodology exploits essential capabilities of the tools in use, and production proven procedures. This paper outlines a step-by-step guide to port an existing UVM-e testbench to SA. The verification user community can use this paper as a template to plan their migration from simulation to hardware acceleration.

I. Introduction

RTL simulation has been the major work horse since the dawn of verification. However, as the designs continue to follow Moore' law, long simulation time is one of the major bottlenecks in verification. Ultra long testcases running over several days or even weeks impact the productivity of the verification team. Many technologies have been introduced to speed up simulation times, and the one that deliveries the most speed up is the use of hardware acceleration box (the box). Hardware acceleration technology has been around for almost a decade. It started out as a niche technology trying to replace custom FPGA emulation boards and today it has become one of the essential verification tools. The top three EDA vendors have their propriety hardware acceleration solutions, and all of them deliver the performance of tens of thousands if not millions times speed up compared to plain old RTL simulation. Traditionally, verification teams use the hardware acceleration box in in-circuit emulation (ICE) mode. In ICE mode, the box behaves like a FPGA emulation board, where the design is compiled into the box and the traffic stimulus is generated by using actual equipment or testers connecting to the box via the speed bridge. ICE-mode can run up to almost at-speed as running the actual design in the silicon. It is ideal for software/firmware testing or performance stress tests. However, due to the limitation of the speed bridge, ICE mode lacks fine control of the traffic stimulus, and it has poor observability in the traffic monitor to debug low level RTL bugs.

To address this problem, the next evolution of hardware acceleration comes in the form of the embedded testbench. Both the design and a synthesizable testbench are compiled into the box; traffic stimulus is generated from the embedded testbench running inside the box instead coming from the speed bridge. The compiled design and testbench run as fast as ICE-mode, and the embedded testbench provides full control over the traffic stimulus. However, synthesizable testbenches are very difficult and time consuming to implement, and it is often as complicated, if not more, as the RTL itself. Embedded testbenches also lack the convenience and ease of use of being implemented using a high level verification languages (HVL), such as Specman *e* or SystemVerilog (SV). Writing embedded testbench is like writing a piece of RTL to test another piece of RTL. Finally, the next logical evolution in hardware acceleration is simulation acceleration (SA) which combines the best of the both worlds, the speed of the hardware accelerator and the usability of a HVL testbench. In SA mode, the design is compiled and run inside the box, while the HVL testbench is running in the simulator connected to the box. In SA mode, the design is running much faster than the testbench that its run time is negligible compared to the run time of the testbench. The simulation speed is no longer limited by the speed of the RTL simulation; the simulation runs as fast as the testbench runs plus a small transaction overhead from communication between the box and the testbench. In short, SA mode is like running the RTL simulation in a superfast computer.

PMC is one of the early adopters of SA. We started experimenting with SA in 2011 when Cadence first announced SA support in their Palladium platform. For a trial project, we selected a complex, mixed-language RTL design (Verilog, VHDL and SystemVerilog) coupled with a state-of-the-art UVM*e* Specman testbench that supports transaction level processing. The initial bring-up of this design/testbench in Palladium running in SA mode took us over a year. Enhancing the testbench to run in SA mode is a very steep learning curve; it involves an in-depth knowledge of HVL, writing synthesizable RTL code and low-level C language programming. Additionally, a relatively new tool chain presented an additional hurdle. At the end, we demonstrated a 40x speed up for a 26M gates design with heavily patched RTL [1]. Back then, porting a testbench to SA seemed like black magic, the process wasn't scalable and difficult to transfer the process to other projects. Flash forward to 2014, after several iterations of the tool and a few more designs ported to run in SA mode, we have refined our methodology on porting an existing testbench to run in SA mode. The time it takes to get a testbench up and running in SA mode is drastically reduced from over a year to just a few months or even merely a couple of weeks. Finally SA is ready for widespread adoption and it is easy to pick up by any verification team by using our methodology.

In this paper, we are presenting a methodology of porting an existing UVM-e testbench to run in Palladium SA. This paper is structured as a step by step guide on how to migrate the testbench. The first section discusses the prerequisite of the testbench, what types of testcases are best suited to run in SA. The following sections go over types of simulations it takes to debug the migration, the RTL compilation flow, the testbench enhancements, and regression management strategy respectively. We shared lessons learnt and pitfalls users should be aware of. Finally, the paper presents some benchmark results for reference, followed by a brief discussion on the future development of this methodology and concluding remarks. The verification user community can use this paper as a template to plan their migration from simulation to hardware acceleration.

II. TESTBENCH PREREQUISITES

Not every testbench or testcase is suitable to run in SA mode. Previous papers [2][3][4] highlight some important points and example calculation to help users determine whether a testbench is suitable to run in SA mode. In this paper, we are focusing on porting the testbench for transaction base acceleration running in blocking mode, where the testbench and the hardware are running alternately. We have experimented with signal based acceleration in the very beginning of the trial; although it sounds like the logical step to start hardware acceleration, it turned out to be a dead end. Signal based acceleration could not deliver the simulation speed up performance to justify the cost of the hardware accelerator. We also investigated using non-blocking mode where time advances concurrently instead of alternately for both the software and hardware domains. In theory, non-blocking mode could deliver the best performance, since the testbench and the hardware are running in parallel, but it changes the behavior of the simulation testbench which we saw as undesirable.

The primary performance bottleneck of SA is the execution time of the testbench. It is recommended to profile a typical simulation run and make sure the testbench uses less than 2-3% of the total CPU time in order to achieve a meaningful speed up in SA mode. Smaller design tends to yield higher testbench CPU usage, so they are not suitable for SA in general. Testbenches that are structured to use excessive interactions between the testbench and the DUT, such as continuous monitoring of RTL signals on every clock, are not well suited for SA acceleration. The user should investigate whether those interaction are required for the testcases targeted for SA mode and see whether those interactions can be disabled or minimized in the testbench. Some testbench CPU usage is simply an artifact of poor coding style, and thus the user should optimize their code to yield better simulation performance. Techniques for testbench optimization is outside the scope of this paper; user can refer to [5][6] for more details.

Another potential performance bottleneck is the synchronization and data transfer between the testbench and the box, but its performance impact is relatively low compared to the CPU runtime of the testbench and the context switch overhead between software and hardware. The testbench is running on a host simulator which is directly connected to the box by an ultra-high bandwidth cable. The throughput of the cable is very high; it is rarely a bottleneck when transferring big chunks of data between the testbench and the box. Therefore as long as the traffic stimulus can be generated by the testbench without taking up too many CPU cycles, there is no need to generate the traffic stimulus form a synthesizable BFM inside the box. Moreover, writing a synthesizable BFM for data generation has the same challenges as writing an embedded testbench, which is complicated and time consuming to write. It is more productivity to reuse the traffic generator VIP from the simulation testbench given that the traffic generators support transaction level processing. The synthesizable BFM and collector inside the box simply act as dumb pipes to transfer data between the testbench and the box, and most of the data processing reuses existing code in the testbench.

The last potential performance bottleneck is the time it takes to download the snapshot into the box and to upload the waveforms at the end of simulation. For a very short testcase, the download and upload overhead may exceed the testcase execution time, which results in deceleration instead of acceleration. The ideal testcase candidate for SA should have a long run time, with long periods of minimal interaction between the testbench and the design other than sending and receiving bulk traffic with occasional event based signal monitoring. It is fine for the testcase to have heavy interaction between the testbench and the box in a given test phase that contributes to a very small percentage of the total simulation run time. The speed up factor during that test phase will slow down significantly, but the testcase still maintains a decent overall speed up.

III. Types of Simulations

The migration of a simulation testbench to run in SA mode is a multiple step process. It is almost impossible for anyone to implement all the required testbench enhancements and the have the SA simulation working the first time it is brought up in the box. Due to the high cost of the hardware accelerator, it is often shared among many verification teams; thus, it is not economical to lock down domains in the box to debug SA integration problem interactively in live sessions. It is easier to break down the migration process into four phases using different types of simulations; each has its strength to identify different kinds of problems.

1) Normal simulation. It is recommended to use the same simulator from the same vendor of the hardware accelerator. There are many subtle differences in how the RTL behaves in different simulators; it is hard to guarantee tools from different venders will interoperate correctly. First, select a testcase as the target of the first bring up. This testcase should not be too long, which allows many quick debug iterations, but it should not be too short so it can provide some interesting performance metrics. This testcase should avoid using backdoor register accesses and disable all unnecessary interaction between the testbench and the DUT. For this step, run the testcase until the end of the simulation, save the seed and the log file, which will be used as the golden reference in the following phases. If it is not already done, the user should also run profiling to identify and fix performance bottle neck in the testbench.

2) SA_SIM simulation. This type of simulation is used to debug errors in the SCEMI pipe and DPI integration of the testbench. The DUT is still running in the RTL simulator, but the testbench is fully enhanced to run in SA mode. The testbench should not consume any simulation time, it should identify and disable all external port bindings and be recompiled with a clean stub file. Re-run the selected testcase with the saved seed and it should have the same simulation result as the saved log file from previous phase. Identify and fix any discrepancy in the simulation result, which is probably due to a mismatch in the behavior of the transaction BFM and collector. Users can also rerun profiling to benchmark the performance of the transaction BFM and collector. Usually the SA_SIM simulation runs slightly faster than normal simulation.

3) SA_SW simulation. This type of simulation uses the host simulator to run both the testbench and the DUT. The DUT is synthesized to the binary format of the hardware accelerator and is simulated using a model of the hardware accelerator which runs inside the host simulator. Again, rerun the selected testcase with the saved seed and compare the simulation results. This step is to identify any potential bugs in the tool flow that synthesized the RTL into the box. Usually SA_SW simulation is somewhat slower than normal simulation due to the overhead of the software model of the box. Since the launch command is identical to the actual SA simulation, it is also used to debug the shell script environment that runs the testcase.

4) SA_HW simulation. The testbench is running in the host simulator and the DUT is running in the box. First, run SA_HW in tbrun mode. Here, the box is running in lock step with the SA_SW simulation which flags any potential mismatches between the software model of the box and the actual hardware implementation of the box. Finally, run the actual SA simulation using SA_HW in normal run mode, where user can measure and benchmark the speed up factor of simulation acceleration.

IV. COMPILE THE RTL FOR THE HARDWARE ACCELERATOR

Based on our experience, the most challenging part of the SA migration process is compiling a complex mixed language design implemented in Verilog, VHDL and SystemVerilog into the hardware accelerator box. Although all the compilation tools are from the same vendor, the SA tool chain is not exactly the same as the commonly used tool chain in frontend simulation and backend RTL synthesis. The Verilog, VHDL and SV parser in the SA tool chain may have a slightly different implementation and have a different interpretation of certain syntax of the code, which results in mismatching behavior between SA and normal simulation. The frontend simulation tool chain and backend RTL synthesis tool chain are more mature comparing to the new SA tool chain. It is highly recommended that the user starts a trial RTL compilation targeting the accelerator early in the project, before spending considerable efforts in testbench enhancement to support SA. If the RTL cannot compile for the box, then there is zero speed up. Although there are occasional compilation issues with the SA tool chain, there are always workarounds available to patch the RTL code to avoid those issues. However, we do not recommend that the user branches off from the original RTL code and maintain a patch copy of the code just for SA. It will be a maintenance nightmare to keep track of the patches and on-going changes in the design. Moreover, it introduces a verification gap as we cannot guarantee that the patched code behaves exactly the same as the taped-out version of the RTL. When the user encounters a tool issues, it is recommended that the user first confirms that the same piece of code works fine in the frontend and backend tool chain, then the user should report the problem to the tool vender. If possible, the user should also send a tar ball of code snippet to help the tool vender recreate the error for internal debug. Once the vender is able to confirm the tool bug, usually it takes a few days for them to distribute a new patch of the SA tool chain to fix the problem.

Given that there is no tool problems, getting the design to compile for SA is fairly straight forward. The SA compilation flow can reuse most of the compile scripts and reuse components from the ICE compilation flow. We recommend that the user uses the normal simulation workspace for SA compilation instead of creating separate workspaces and vaults. There are many benefits of using the same workspace, which includes incorporating design changes, since there is only a single RTL code base to update and debug. The user can launch a testcase in either normal simulation or SA mode from the same workspace and use the most appropriate simulation type to debug RTL problems. Assuming RTL compile scripts for normal simulation is readily available; compiling the RTL for SA requires the following steps: (our example is illustrated using the Palladium compile flow)

1) Define the path to the compiled library for SA. (Generate a **`libmaprc`** file from the existing **`cds.lib`** file). It is recommended to put the SA compile library in the same path as the simulation compile library. This allows easier cleanup of the workspace when checking-out updates of the latest RTL changes and it also allows partially recompile of the design by cleaning up some of the compiled libraries.
2) Generate synthesizable RAM models for all behavioral RAM models in the design. This is the same RAM generation script used in ICE mode.
3) Identify all non-synthesizable common behavioral components in the design, such as RAM models, DesignWare or gtech libraries, and then swap in a synthesizable version. There are many ways to swap in the synthesizable code, we recommend using SV configuration to determine the binding at the elaboration phase for different snapshots. This allows both non-synthesizable and synthesizable versions of the compiled design to co-exist in the same workspace. The user will compile all the synthesizable version of the code to be swapped into an **`accel_lib`** library and set it as the default binding in the SV configuration file with

```
config accel_top;
      design work_lib.top;
      default liblist accel_lib;
end config
```

4) Parse the simulation compile scripts; replace all references of the simulator compiler with the SA compiler. (Replace **`ncvhdl`** with **`vhan`**, **`ncvlog`** with **`vlan)`** and add **`+rtlCommentPragma`** and any other **`synthesize_on/off`** pragma to compile the synthesizable version of the code.
5) After the RTL code is analyzed, call the synthesizer (**`ixcom)`** to synthesize the RTL code into the accelerator binary. Since everything in the design should be synthesizable and target to run in the hardware, it is more convenient to force the tool to synthesize everything into the box. (**`-rtlchk off`** option in **`ixcom).`** It avoids the synthesizer to guess which module is synthesizable and which one should runs in software module, as this

process is prone to error and may end up with having some module running in software instead of hardware, which will heavily impact the SA performance. Instead, the tool simply reports an error when it encounters non-synthesizable modules. This step may require a few iterations to clean up all the non-synthesizable code in the design. Unlike the binary for ICE mode, the SA binary are not limited by the physical location of the speed bridge. Thus, it can run in any domains or boards in the box. The user should specify the design to target contiguous domains in the box and enable the symmetric boards option to allow the design to be deployed across any domain. If the SA binary generation is successful, the user can find out how fast the design runs and the number of domains required from log files, like

```
INFO (db2util-1067): This design is scheduled in 400 steps.
INFO (qt2dadb-1086): Maximum emulator operating speed is 1238 kHz.
...
Info        [FLOW_SUCCESS]
Successfully built target (xeCompile)
...
...
00 ET5NLOPT | Assuming instruction usage 80%, the design requires at least 6 domains.
```

V. TESTBENCH ENHANCEMENT

Provided that the testbench already meets the prerequisites described in section 2, there are not many testbench specific changes required to support SA mode, other than extending the VIP to support TBA mode and hook up the reuse testbench controller (TB_CTRL) module that facilitates the interaction between the testbench and the box. The VIP components and the TB_CTRL unit are reuse components that can be shared among different verification projects. Once they are already created, it significantly lowers the barrier to port additional testbenches to support SA mode. In fact, in our SA methodology, project specified code to support SA mode in each specific testbench is limited to a single e file with just a few dozen lines of code. We also highly recommend maintaining the same code base for both normal simulation and SA simulation. Thanks to Specman's unique aspect oriented programming (AOP) ability, which allows different implementation of the same function API based on the aspect configuration at run time, we can use the UVM sequence code verbatim in SA mode. The exact same testcase importing the exact same UVM sequences runs on both normal simulation and SA and will generate the same simulation result with the same seed. The testbench supports running the testcase in two different aspects, one for normal simulation and one for SA mode. The user has to refactor the code to consolidate external port binding and agent type under the normal simulation aspect, so none of the external ports is bound when running in the SA aspect. The following is an example of how to define the two aspects:

```
type uvm_abstraction_t : [UVM_SIGNAL, UVM_ACCEL];
extend any_uint {
      uvm_abstraction : uvm_abstraction_t;
      // pass the aspect down the unit tree
      keep uvm_abstraction == get_parent_unit().uvm_abstraction;
};

// disable the external binding in UVM_ACCEL
extend UVM_SIGNAL <unit name> {
    keep agent() == "NCSV";
      keep bind(<signal port>, external);
};
extend UVM_ACCEL <unit name> {
      keep agent() == "";
      keep bind(<signal port>, empty);
};
```

Enhancing a 3rd party VIP to support SA mode is probably the biggest challenge in migration. Unless the VIP comes directly from the vendor of the hardware accelerator, it is very hard to find ready-made accelerable VIP in the market. For in-house reuse VIP and testbench specific functions, [7] provides a detail description on different communication methods between the testbench and the box. We are not going covering the implementation of those methods in this paper; please refer to the corresponding document for detailed information. Instead, we include a brief discussion on how we use each of those communication methods in our SA methodology; users are welcome to use the information for reference. Please keep in mind that there is no "one size fits all" solution in selecting which communication method to use as it highly depends on the characteristics of the targeted application.

1) SCE-MI pipe. This is the preferred method to transfer high data rate traffic between the testbench and the box. The SCE-MI pipe proxy built into the SA tool supports auxiliary functions, such as query queue status, flush queue, etc. The full Accellera SCE-MI specification is very complicated and it is probably overkill for VIP TBA mode operation. Here is an analogy to help the user to understand SCE-MI pipe concept: On the e side, the SCE-MI is like a blocking TLM get or put port. It supports auto pack/unpack the physical field of a struct, the struct is packed into a 2 dimension byte array with the size of the pipe width X pipe depth. On the SV side, it is like one side of a RTL FIFO module. On each clock cycle, the TX side reads one word with the size of the pipe width from the FIFO, and the RX side writes one word into the FIFO.
2) e/SV DPI interface. This is the preferred method to implement a command/control interface in the testbench and VIP. It works like a method port, allowing e to call a SV function and vice versa. It is required to write some C glue to stitch the e/SV DPI call. The DPI interface supports non-TCM function calls only, however TCM calls can be implemented using two separate DPI calls under the hood, first setup the forward call from e to SV, then wait in the box until the timer expires, then trigger a backwards call from SV to e. Any DPI call will trigger a context swap between the testbench and the box; consequently, too frequent DPI calls will impact performance.
3) MARG (direct memory copy) interface. This is the preferred method to transfer a memory block between the testbench and the box in zero time. SCE-MI pipes are implemented using MARG under the hood. MARG provides raw memory access to the box, it is the fastest communication method, but it does not provide any helper function to facility the memory transfer. It is good for one-off deposits or read back a value from the deposit. It is also the mechanism to implement custom pipes if the SCE-MI pipe does not meet the VIP requirement. It works on any synthesizable Verilog array or register type. For Verilog array, users have to declare a piece of memory in the testbench with the same size of the memory array in the box. For register types, users have to explicitly export the register signal in the HDL code. In the testbench, the user has to facilitate when to wake up the testbench and call the MARG memory copy methods to transfer the content of one memory to another.
4) Tcl deposit/force/value. Believe it or not, Specman can still use **`simulator_command()`** to issue simulator tcl command in SA mode. The testbench can still use tcl deposit/force/value commands to communicate with the DUT. This method is very slow, but it is a very handy debug method as a last resort, since Specman can extend the testcase and inject additional code without recompiling the SA snapshot or even without restarting the simulator.

The TB_CTRL module is a reuse component that encapsulates commonly used functions to enable rapid testbench migration in SA mode. It facilitates the interaction between the testbench and the box; it has the following major features implemented using the DPI interface:

1) Emit a Specman tick event periodically based upon a hardware clock running in the box. It is used to wake up the testbench and move the time consuming sequences forward in simulation time.
2) Provides a **`wait_delay()`** method replacing the standard Specman **`wait delay`** statament. In UVM_SIGNAL the **`wait_delay()`** is just a wrapper for **`wait delay`**, but in UVM_ACCEL, **`wait_delay()`** converts the time notation into counting the number of elapsed tick events or the nearest HW clock cycles in the box depending on the timer granularity configuration.
3) Provides general purpose output (GPO) pins to drive a simple repeating pattern to a pin using the TB_CTRL module in the box via a method call from the testbench. Since there is no external signal port binding in the testbench in SA mode, any output to external signal port should be rerouted through the GPO pins of the TB_CTRL module. The number of the GPO pins is configured by SV parameter of the TB_CTRL module. The testbench reset in SA mode is also implemented using the GPO pins.
4) Provides general purpose input (GPI) pins that will emit an event to the testbench if there is any change to the pin and/or the new value matches the predefined mask value. Any input from external event port should be rerouted through the GPI pins of the TB_CTRL module. The event will cause context swap between the box and the testbench and can be used to trigger testbench function or UVM sequence like normal simulation. Interrupt monitoring of the DUT is implemented using the GPI pins. In normal simulation, the testbench may monitor all interrupt changes, but in SA mode, the interrupt should be configured to flag only serious design failures that require intervention from the testbench.

Very often, many verifiers work on the testbench, writing testcases and UVM sequences. The person who is responsible for the testbench migration may not know all instances of code violation for SA, such as using tick notation, orphaned external bindings and wait delay statements. We have implemented the following debug utility to help the user inspect the testbench code for code violation using Specman macros and the reflective interface.

```
define <delay_override'action> "wait delay(<exp>)" as {
    outf("WARNING wait delay at %s\n", source_location());
};
define <force_override'action> "force <any>" as {
    outf("WARNING force at %s\n", source_location());
};
define <tick_read_override'exp> "'<any>'" as {
    outf("WARNING tick read at %s\n", source_location());
};
define <tick_write_override'action> "'<lval'any>'=<rval'any>" as {
    outf("WARNING tick write at %s\n", source_location());
};
extend any_unit {
    check_generation() is also {
        if (agent() != "") {
            outf(WARNING agent() at %s is not empty\n", e_path());
        };
        var s       : rf_struct = rf_manager.get_struct_of_instance(me);
        var f_list  : list of rf_field = s.get_fields();
        for each (f) in f_list {
            var t : rf_type = f.get_type();
            if (t.get_qualified_name() == "pm_mtf::uvm_abstraction_level_t") {
                var v :int = f.get_value(me).get_value().unsafe();
                if (t.as_a(rf_enum).get_item_by_value(v).get_name() != "UVM_ACCEL") {
                   outf("WARNING uvm_abstraction_level_t at %s.%s is not UVM_ACCEL\n",
                   e_path(), f.get_name());
                };
            };
        };
    };
};
extend sys {
   check_generation() is also {
        for each (p) in sys.get_ports_recursively() {
            if bind(p, external) {
                outf("WARNING Port %s is bound external\n", p.e_path());
            };
        };
    };
};
```

VI. REGRESSION MANAGEMENT

Launching testcases in normal simulation and in SA mode uses the exact same interface, the only difference is the simulation snapshot loaded into ncsim and the targeted machine to run the job. Our regression manager (`vManager`) is used to facilitate regression runs in SA mode, just like the regressions running in normal simulation. Due to the overhead of downloading the design into the box, it is recommended to group all testcases using the same snapshot into one regression session. The pre session script should reserve the domains required by the design in the box, then it should use the keep host alive feature of the tool to keep the design in the box between SA simulation runs to avoid having to download the design into the box again. The post session script should free the reserved domains once the regression suite is finished.

When a testcase fails in the regression suite, even though recreating the failure using SA simulation is considerably faster than normal simulation, it is still more convenient if the regression manager re-launches the failed testcase automatically and has the simulation session stop right before the failure point, waiting for the verifier to debug at their next opportunity. However there is one concern in auto re-launching for SA simulation, the hardware accelerator has a very high cost, and thus it is unwise to leave some domains sitting idle for the whole night waiting for the simulation to be debugged. When the simulation session stops at the failure point, it should hot-swap from SA_HW mode into SA_SW mode and release the domains, thus allowing other testcases to utilize these limited resource in the box.

VII. Benchmark

Table 1 presents some benchmark results for reference. Project A was the initial trial run, we spent over a year learning the fundamental of SA and struggling with various tool problems. Project B was the first project to utilize the lessons we have learnt in project A, where we refined our SA methodology and prepared it for wide adoption. The schedule for project B includes the time to build all the SA compilation scripts, TB_CTRL reuse components, porting two additional VIP to support SA and deal with occasional tool problems. Project C applied the matured SA methodology; it neither has any new VIP SA mode development nor encounters any tool problems.

TABLE I - Benchmark Results

	Project A (2012)	Project B (2014)	Project C (2014)
Design size (in Mgates)	26	38	50
Domains used	6	6	8
Speedup factor	40x	67x	52x
Migration schedule	1 year plus	3 months	3 weeks

VIII. Future Development

One of the biggest drawbacks of SA is that it does not support checkpoint save and restore like normal simulation. Saving a simulation checkpoint and then restoring it to the closest simulation time before the point of failure drastically decreases the debug turnaround time. No matter how fast SA runs, it is always slower than simply reloading a checkpoint saved right before the point of failure. We will work with the tool vendor to implement checkpoint save and restore in the next evolution of the SA methodology.

IX. Conclusion

Compared to the benchmark results in [2][3], our speed up factors do not seem very impressive. We focused our efforts in bringing up SA mode and put it to use in the shortest period of time. We did not focus on optimizing the testbench to deliver the best possible speed up factor, other than fixing some obvious performance bottleneck due to poor coding, such as list operations that keep allocating and releasing memory. This is a calculated trade off based on the expected use model of SA. No matter how hard we optimize the testbench, SA can never run nearly as fast as ICE mode. For heavy duty simulations that require ultra-long simulation times, one should always stay with ICE mode. The advantage of SA mode is to shorten the normal simulation time while keeping the flexibility of the HVL testbench. As long as the simulation runs fast enough, like a simulation finishes within an hour, it does not provide much extra value to squeeze the simulation time down to thirty minutes. We have done some internal ROI calculation, factoring in the CPU operation cost and license fee of normal simulation. We found that the breakeven point for SA is around 30-40x depending on the discount of the license fee and the lease contract of the hardware accelerator. As long as the speed up yields a positive ROI, it is a sound investment to use SA mode. On top of the calculated ROI based on the raw simulation throughput, there are other intangible benefits like decreased debug turnaround time and increased productivity of the verification team.

Acknowledgment

I would like to thank my colleagues in PMC's CAD department and the helpful FAEs in Cadence for their support in bring up the SA simulation environment.

References

[1] H. Chan, J. Huang and D. Allen, "Functional Verificaiton of next Generation IC's with Next Generation Tools", CDNLive, 2012

[2] S.Kumar, G.Kumar, P. Kulkarni and V. Rao, "Accelerating UVM Testbenches to Achieve 100x Simulation Performance", CDNLive, 2013

[3] V. Rao, K. Kumar, V. Verma, G. Kumar, "Using Simulation Acceleration to achieve 100x performance improvement with UVM based testbenches", DVCon India Proceedings, 2014

[4] K.A. Meade, "UVM Testbench Considerations for Acceleration", DVCon Proceedings, 2014

[5] E. Shneydor, "Performance-Aware e Coding Guidelines Series Part 1-5", Cadence blog (http://community.cadence.com), 2009

[6] F.Kampf, J.Sprague, and A.Shere, "Yikes! Why is My SystemVerilog Testbench So Sloooow?" DVCon Proceedings, 2012

[7] S. Aggarwal, "Boost Efficiency and Performance of Simulation Acceleration Through New Rapid Adoption", Cadence blog (http://community.cadence.com), 2014