SOC Simulation Performance – Take a second look at your VCS setup

Kendall Chan

AMD
Markham, Canada

www.amd.com

ABSTRACT

Synopsys is always developing new options to address VCS compile and simulation performance concerns. New technologies like native-low-power, multicore, partition compile, etc... tend to provide some incremental benefit but they also tend to require some commitment to support their deployment and/or development. This paper reviews a few 'old school' VCS methods for improving simulation performance that have been re-explored in AMD in the last few years. Synopsys VCS has always had many optional settings and methodologies that affect simulation performance that are easily overlooked by many SOC teams. Many of these can be relatively simple to deploy and provide significant benefit. Taking a fresh look at plain old VCS is always a good exercise for any SOC ASIC team.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

The list of options VCS supports can easily overload most users. This paper hopes to give a second look to some legacy options a verification engineer should review to help improve VCS simulation performance for an SOC environment. Many of these have been available for some time, but they are often overlooked.

We will start with a quick background on a recommendation for running simulation benchmarks. Given that not every option necessarily provides a benefit, having a proper benchmark setup that produces clear results provides clarity for decision making. From there we will go through a small list of some VCS simulation performance strategies. It is the hope of the author that you can identify a few things that you can apply to your own environment.

Options to improve your simulation performance often fall into one of these buckets:

- VCS Options - managing VCS simulation options so that unused simulation support is disabled when not needed. Most options are about managing tradeoffs – you basically make decisions to disable features of the simulator that you can do without for a group of tests. The verification team needs to be aware to enable things when they are needed.
- Your Env Settings - make sure you configure your testbench environment appropriately (for example making your clock settings optimal for minimizing runtime, or not running with many extraneous verification checkers enabled).
- Coding Efficiency issues – VCS is an event based simulator, so code efficiency is a concern in areas of the environment that are heavily triggered. Profiling through VCS and attacking the code areas highlighted is the basic approach users can follow here.
- Minimizing blocks of gates in your RTL sims. There are often cases in an SOC environment where some IP team pushes to run some portion of their code as gates. Unless this IP is very small, the simulation performance effects should be reviewed before going down this path.
- Simulate less – simply minimize what you run. For example, you should run what you can as a STUB or BFM instead of real RTL, as required for your test. This is typically the best option to improve your performance but this requires careful planning with the verification team. This also creates more different configurations/builds of your design which will require more diskspace.

The latter four topics are typically the best areas to improve your performance, but they are also very tied to your specific design and testbench requirements. The general topics we will discuss here will focus on the first item.

Apart from the quick review of benchmarking, this paper will focus on the following few options all SOC teams should explore for optimizations. We have reviewed these on a graphics design at AMD and some have been pushed for deployment across various AMD SOC environments. In each case, I'll review the option, a suggested use model, an example benefit, and the tradeoffs. With this collection of options used together, you could get some substantial benefits to your simulation performance for very little cost and effort.

- Profiling : VCS Option "–simprofile" (see Section 3 - "Time Profiling")
- Top Levels: VCS Option "–top " (see Section 4 - "Managing your top level modules")
- Assertions: VCS Option "–assert disable_cover" (see Section 5 -"-assert disable_cover")
- Assertions : $assertoff (see Section 6 - "$assertoff")
- PLI Access: VCS Option "+vcs+learn+pli" (see Section 7 - "Limit your PLI access")
- FSDB performance: "setenv NOVAS_FSDB_SKIP_CELL_INSTANCE 1" (see Section 8 - "setenv NOVAS_FSDB_SKIP_CELL_INSTANCE 1")
- Assertions: VCS Option "–assert novpi" (see Section 9 – "-assert novpi")
- VCS debug options too expensive to enable by default (see Section 10 – 'VCS "debug" options too expensive to enable by default')

The examples and related benchmark results provided below were reviewed on AMD graphics IP designs using VCS 2012.09-SP1-1. All the benchmark simulations are using a single basic graphics test. For areas related to waveform dumping, note that Verdi 2013.04 was used.

---

2. A Benchmarking Strategy Recommendation

No sim performance discussion is really started without a proper benchmark setup to measure the benefit. While benchmarking should be as simple as running a reproducible test on a dedicated machine, it can actually turn into a complex problem as will be explained. Not every VCS option provides a clear benefit and a proper benchmark helps identify the tradeoffs and will assist in making clear decisions on what options should be used.

Running proper benchmarks actually posed a bit of a puzzle for our work at AMD. The problems for us fell into these areas:

1. A completely randomized environment – we could rerun test sequences on the same design release/changelist with a consistent seed but there was no option to run the same test sequence across different design releases. Management is typically interested in tracking simulation performance trends for the project over time. In this regard you need a reproducible testcase across subsequent version updates of your design.
2. The main hosts to run benchmarks on SOC environments were AMD Opteron Multicore servers. We had some strange behaviour that took some time to investigate.

A proper benchmark setup is also a key platform to create regular datapoints to plot performance trends over time. Earlier AMD SOC teams focused on tracking performance trends only across large regressions. This isn't a wrong strategy, but the results typically come too late for the teams to analyze and react to and it is more of use to high level management than to the project teams. It is the recommendation of the author that it is much better for the teams to focus on benchmarking and plotting the trends for a few representative tests.

Of the two issues listed above, the second item will be described here as the same issue may be applicable to other teams also running on AMD Opteron servers.

How to run your benchmark tests on your dedicated host:

Once you can get your hands on a dedicated system you might be a little surprised with your benchmarking results. Here is an example of a set of successive test results running on a dedicated AMD Opteron Multicore Server (8 x quad-core = 32 cores) machine. Each test case was run in isolation without any other simulation processes running. Even in this case where we are running the same testcase in the same workspace with no changes being introduced we noticed a big variation over the results.

```
CPU Time: 10786.260 seconds = 02:59:46
CPU Time: 11465.730 seconds = 03:11:05
CPU Time: 11041.730 seconds = 03:04:01
CPU Time: 10768.630 seconds = 02:59:28
CPU Time: 10937.980 seconds = 03:02:17
CPU Time: 10824.110 seconds = 03:00:24
CPU Time: 10747.610 seconds = 02:59:07
```

The variance for the above data is 65030.691. This observation of large variations between the runs was very consistent across hosts and sites in AMD where similar tests were run. This actually stumped most of our IT folks. Someone in AMD eventually helped to identify a linux command to solve this riddle – we should submit our simulation commands through the `numactl`[1] linux command. Using `numactl` we are able to run our simulation processes with consistent processor and shared memory placements. Now we start to see the more consistent results we expect.

```
numactl –physcpubind=0 simv +vcs+lic+wait +foo_option=1 ...
```

Here are some recent results run using `numactl`,

```
CPU Time: 10858.700 seconds = 03:00:58
CPU Time: 10896.140 seconds = 03:01:36
CPU Time: 10893.200 seconds = 03:01:33
CPU Time: 10903.970 seconds = 03:01:43
CPU Time: 10885.990 seconds = 03:01:25
CPU Time: 10882.930 seconds = 03:01:22
CPU Time: 10894.810 seconds = 03:01:34
```

We now see more consistent runtimes and this makes running benchmarks much easier. The variance for this new set of data is 213.824. With the use of '`numactl`' in our benchmark setup on the same testcase, we can achieve < 2% variation run-to-run and typically < 1%. Many simulation performance options are very incremental so having better visibility to small incremental improvements helps to grade the options we are testing.

When running simulation benchmarks you should track both simulation cycles (time in 'ns' simulated) and VCS "CPU Time". The former helps to confirm you are running the same test se-

[1] '`numactl`' – Linux command to control NUMA policy for processes or shared memory - for more details you can refer here: http://linux.die.net/man/8/numactl

quence for each test run iteration. Another suggestion that is very helpful is to keep your source files and output on local disk (/tmp) of your benchmarking host to avoid variability on the network fileservers – however, in many cases at AMD we cannot do this due to the requirement that files must be kept on secure filesystems.

The next group of sections will be a review of a few selected VCS Options:

---

3. VCS option "-simprofile" – Time Profiling

Option:

The most basic option in VCS for reviewing simulation performance is time profiling – this involves compile time option `-simprofile` and runtime time option `-simprofile time`, as well as the "`profrpt`" command following simulation.

Compile time :

```
vcs -simprofile <your other compile options>
```

Simulation run time:

```
simv -simprofile time <your other runtime options>
```

Post simulation:

```
profrpt -view time_all -output time.dir simprofile_dir/
```

Use Model:

Using the `-simprofile` options for compile and runtime will create a runtime database. This database after simulation is processed through a "`profrpt`" command. The output is a set of html page views that help to highlight areas taking the most time overhead. This includes a summary view, module view, instance view, PLI/DPI/DirectC view and construct view.

Teams should create time profiles on a regular weekly interval. As the design comes together over weeks and months, simulation performance problems often occur. It is handy to have current and past time profiles available for reference to review.

Some suggestions on using the different areas of the html report:

1. The "Time Module" view is probably the most useful view. It helps to rank the top modules that take up the most simulation time. From our experience reviewing time profiles in AMD, anything over 2% should be reviewed carefully. The report also shows the instance count of modules so that should be taken into account as well when reviewing the results.

   A sample "Time Module" view is shown below:

Time Module View

Module	Inclusive Time	Percentage	Exclusive Time	Percentage
onion_ovc_pkg	434.84 s	5.52 %	434.84 s	5.52 %
garlic_if	224.49 s	2.85 %	224.49 s	2.85 %
vcs_paramclassrepository	202.09 s	2.57 %	202.09 s	2.57 %
NROSTAHA1_imux4xss2ur	195.33 s	2.48 %	195.33 s	2.48 %
ovl_next	--	--	115.07 s	1.46 %
hdcklban2xss6ur	107.24 s	1.36 %	107.24 s	1.36 %
base_onion_if	102.46 s	1.30 %	102.46 s	1.30 %
cover_active	100.70 s	1.28 %	100.70 s	1.28 %
cgtt_local_dcg_cov	97.07 s	1.23 %	97.07 s	1.23 %
wr_flops	85.09 s	1.08 %	85.09 s	1.08 %
VIRL_SDN_MSDPRB_1	83.21 s	1.06 %	83.21 s	1.06 %
CDCBUFENCLK	--	--	73.65 s	0.93 %
cgtt_local_rcg_cov	72.40 s	0.92 %	72.40 s	0.92 %
esram_bist_pipeline1	71.87 s	0.91 %	71.87 s	0.91 %
NROSTAHA1_sync2msfqxss1ul	70.31 s	0.89 %	70.31 s	0.89 %
ovm_pkg	65.29 s	0.83 %	65.29 s	0.83 %
sclk_tile_root	56.51 s	0.72 %	56.51 s	0.72 %
garlic_ovc_pkg	54.71 s	0.69 %	54.71 s	0.69 %
mapsmcore_emc	51.30 s	0.65 %	47.34 s	0.60 %
perfmon_file_counter	--	--	42.31 s	0.54 %
total	7877.50 s	100 %	7877.50 s	100 %

Page: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

Figure 1 – Sample Time Profile Module View

Clicking on items in the module view will show additional information for the specific module – such as instance count and source file information.

2. The "Time Construct" view is similar to the module view but it is looking at hardware constructs – for example an always block, continuos assignment or an assertion – and ranking by the most expensive constructs being simulated in your environment. The construct view could help with the following:
   - can highlight a few very expensive assertions in terms of simulation time
   - can identify a particular part of a module that might also appear in the module view to better isolate the performance problem.

A sample "Time Construct" view is shown below:

Time Construct View

Name	Time	Percentage
Always	1579.27 s	14.20 %
NoName	541.09 s	4.87 %
NoName	81.27 s	0.73 %
ContAssign	485.65 s	4.37 %
Port	421.47 s	3.79 %
Task	305.73 s	2.75 %
Function	267.57 s	2.41 %
Gate	110.64 s	1.00 %
CoverGroup	71.82 s	0.65 %
total	3365.99 s	30.27 %

Figure 2 – Sample Time Profile Construct View

Clicking on an item in the construct view will show additional information – such as source file and the relevant line numbers.

3. The "Time Summary" view has a few things to watch:
   - note the component of time taken by the profiler itself. This overhead varies with VCS version, and it can be significant.
   - note the total time spent in PLI. If PLI access goes well beyond say 5% it should be reviewed on what C functions are being triggered and whether

those calls can be minimized. This isn't always bad but it should be reviewed.

- note the breakdown of the verilog overhead and particularly the size of the assertion component. If assertions are too high, identify where they are and usually there can be options for the verification team to choose to disable them in some situations.

A sample "Summary" view is shown below:

Time Summary View

Component	Time	Percentage
VERILOG	6663.75 s	59.93 %
Functional Coverage	71.82 s	0.65 %
Module	5172.24 s	46.52 %
Package	808.37 s	7.27 %
Interface	592.20 s	5.33 %
Function Coverage Kernel	10.47 s	0.09 %
UDP	7.71 s	0.07 %
Program	930.66 ms	0.01 %
KERNEL	2038.05 s	18.33 %
HSIM	1851.45 s	16.65 %
DEBUG	319.08 s	2.87 %
PLI/DPI/DirectC	186.35 s	1.68 %
CONSTRAINT	53.55 s	0.48 %
PROFILE	5.33 s	0.05 %
ASSERTION_KERNEL	924.68 ms	0.01 %
Value Change Dumping	17.95 ms	0.00 %
VPD	17.95 ms	0.00 %
total	11118.50 s	100%

Figure 3 – Sample Time Profile Summary View

4. The "Time PLI/DPI/DirectC View" is a good first look into functions that make up the PLI % which reflects the time spent within the C functions that are interfaced to your verilog environment. If you need a more detailed breakdown of what C sub-functions are actually taking up time in the C domain, other tools like GNU gcc profiling tool '`gprof`' can be used for this type of detailed analysis.

A sample "Time PLI/DPI/DirectC View" is shown below:

Time PLI/DPI/DirectC View

Name	Time	Percentage
PLI	182.29 s	1.64 %
$simtool_wait_int	151.24 s	1.36 %
$hydra_inst	21.37 s	0.19 %
$hydra_inst /proj/cipher_synopsys_nobackup/wendlu/ws1_test_time_profile_odie /out/amd64rh3.0.VCS/common/pub/include/simulate.v:15	21.25 s	0.19 %
$fsdbDumpvars	3.39 s	0.03 %
$fsdbDumpvars /proj/cipher_synopsys_nobackup/wendlu/ws1_test_time_profile_odie /out/amd64rh3.0.VCS/kryptos/library/gc-kryptos/pub/src/verif/harvest_monitor.v:286	3.39 s	0.03 %
$ati_mtrand_int32	1.48 s	0.01 %
$hydra_signal_posedge_event	1.35 s	0.01 %
$garlic_mem_read	1.15 s	0.01 %
$garlic_mem_read /proj/cipher_synopsys_nobackup/wendlu/ws1_test_time_profile_odie /out/amd64rh3.0.VCS/kryptos/library/orb-orb3_4_0/pub/src/verif/models/onion_ovc_extended/src/sv /unb_onion_sequencer.svh:57	1.15 s	0.01 %
DirectC	4.05 s	0.04 %
itrace_clock_dpi	1.07 s	0.01 %
itrace_clock_dpi /proj/cipher_synopsys_nobackup/wendlu/ws1_test_time_profile_odie /out/amd64rh3.0.VCS/kryptos/library/gc-kryptos/pub/src/verif/sq_monitors.sv:176	1.07 s	0.01 %
total	186.35 s	1.68 %

Figure 4 – Sample Time PLI/DPI/DirectC View

Over time as the results are reviewed and analysed, you will settle on a set of profile results that you believe are good. This will be your baseline to compare against if/when future changes make negative or positive effects to your simuilation performance. This will help the verification team to identify potential problems sooner.

---

4. VCS option "-top" – Managing your top level modules

Option:

VCS supports a `-top` compile time option to pass a top level configuration or module into your test environment. This is a way of filtering out unwanted extraneous top levels – thus saving memory and potentially also improving your simulation performance.

Use Model:

Prior to VCS supporting `-top` it relied on implicitly determining the top levels based on the elaboration process. Whatever compiled code that wasn't a library and wasn't instantiated would become a top level module. A lot of old ASIC environments might still rely on this implicit scheme for its top level testbench modules. In SOC environments at AMD the top levels have not been monitored well in the IP and SOC development areas for long periods. The SOC top level lists became very big and their contents were not well understood.

To provide an explicit list of simulation top levels for your environment, you provide one or more `-top` options to VCS. To find your list of desired tops, it is recommended that you start with the full list of top level modules reported at the end of the VCS compile log. Take note that as part of this VCS top level list could be modules that are bound to a sub-level of the design through a `bind` statement. You can filter out modules that have ports dangling and are not bound to sub-level of the design. The remaining list should be kept unless signed off for removal by the verification team.

For performance reasons it would be a good exercise for teams to review what top levels are being reported by VCS at the end of the vcs_compile.log. In most cases extraneous tops have little effect but in the rare case with some checkers it is possible for them to be triggered even when you don't intend.

Benefit:

In AMD we have seen cases where a module thought to be `` `ifdef ``'d removed from our testbench was actually still showing high in the time profile. What happened was that a checker module in the environment was verilog `` `ifdef ``'d removed from the testbench, but the vcs compile list still included the file for compile. It was no longer instantiated in the testbench but it became a top level module and was still actively stimulated given that it used cross module connections to reference clocks and monitor signals in the design. Hence it was hurting our simulation performance when the intention was for it to not be in our environment.

There is a chance that implementing `-top` won't give you much benefit for performance but it is the view of the author that keeping things clean helps to prevent potential performance problems down the road.

Tradeoff:

Some drawbacks of defining explicit tops through `-top`:

1. If the required tops are not managed well in IP areas and SOC areas there are chances that checkers that are top level modules might be left out and leave a verification hole. Deploying –top options needs to be properly coordinated.
2. Many development environments have different testbenches between the top level SOC and the major IP sub-blocks of the design that feed into the SOC. In this case, it is important that if `-top` is deployed in the IP environment, it must be deployed as well into the SOC environment. Only implementing the change in the IP testbench may introduce false top levels when the IP code is delivered to SOC.

---

5. VCS option : "-assert disable_cover"

Option:

VCS provides an option `-assert disable_cover`. Here's a small comment from the VCS reference manual:

`-assert disable_cover`
`To disable assertion coverage, use the -assert disable_cover compilation option. By default, when you use the -cm assert option, VCS enables monitoring your assertions for coverage, and writes an assertion coverage database during simulation.` [1]

From the description it appears to be something users can consider within regressions where they are collecting coverage. What is a bit hidden is that this option will also disable cover properties which are enabled by default for functional coverage collection for all VCS sims.

Use Model:

For simulations that don't have intent for creating coverage reports, the `-assert disable_cover` option should be set as default.

Benefit:

Here are some results using an AMD graphics testcase with VCS 2012.09-SP1-1:

	Data (hr:min:sec)	Average (hr:min:sec)	Comments
Reference	Run1 : 03:15:45 Run 2 : 03:15:44 Run 3 : 03:15:37	03:15:42 (1x)	8.35% gain seen with adding `-assert disable_cover` option
add `-assert disable_cover`	Run 1 : 02:59:36 Run 2 : 02:59:13 Run 3 : 02:59:14	02:59:21 (0.916x)	

Table 1 - Benchmark data for applying "-assert disable_cover"

Tradeoff:

Teams need to make sure their environment has some organized way of detecting when coverage options are not enabled such that you can enable "`-assert disable_cover`".

---

6. System Task $assertoff

Option:

Over the last ten years the use of SVA assertions for functional verification and coverage has increased substantially. At AMD, verification teams have put many thousands of assertions into their environments. Many of the assertions we have at AMD are controlled through a '`disable iff`' construct in the assertion property.

Here is a small sample code of a SVA assertion definition that utilizes the '`disable iff`' construct for an asynchronous reset:

```
module foo ();
   :
   :
  property p1;
    @(posedge clk) disable iff (Reset) b ##1 c;
  endproperty
  assert property (p1);
endmodule
```

This assertion means that if `Reset` becomes true at any time during the evaluation of the sequence (b follows c by 1 clock event), then the attempt for `p1` is a success. Otherwise, the sequence `b ##1 c` must be evaluate to true.

In AMD, there have been cases where sometimes the condition for the '`disable iff`' is actually controlled at the testbench level and dependent on something other than a reset. In this situation typically the condition would be set for the duration of the tests. We have the intention to always disable these assertions for some tests and to always enable for other tests.

```
module foo ();
   :
   :
  property p1;
    @(posedge clk) disable iff (assert_off_enable) b ##1 c;
  endproperty
  assert property (p1);
endmodule
```

The above '`disable iff`' idea works well for cases of reset where you want to disable the assertion for specific short periods. The confusing part of the above is that this

'disable iff' has no benefit to improving your simulation performance when the assertion gets disabled. VCS still processes most of the overhead to support the assertion. So the construct works well if reset was relatively short, but from a sim performance perspective if your reset is very long or if the "disable iff" clause evaluates true for the entire test, this isn't the best option. In places where the assertions account for a large overhead in your environment, it would be good to have this recoded.

In our AMD IP blocks there are a lot of assertions which get inherited into the SOC environment. We noticed that setting the "disable iff" condition to be true did not improve our simulation performance. Synopsys RnD also helped to confirm this observation for us. The only way to tell VCS to stop the activity and improve performance is to manage the enabling and disabling via the $assertoff utility. Here is the description from the 2012.09 VCS reference manual.

```
$assertoff
Tells VCS to stop monitoring any of the specified assertions
that start at a subsequent simulation time. [1]
```

Applying this to our earlier example, we might have something like this:

```
module assert_control ();
   :
   :
  initial begin
    if (foo.assert_off_enable == 1'b1)
      $assertoff (0, foo);
  end
endmodule

module foo ();
   :
   :
  property p1;
    @(posedge clk) disable iff (assert_off_enable) b ##1 c;
  endproperty
  assert property (p1);
endmodule
```

Use Model:

This is a situation where you have assertions accounting for a large portion of your simulation overhead and where you want to disable the assertions for some test cases. In a large SOC environment, for test cases where you want to hierarchically disable some assertions for specific build configurations, define some part of your top level testbench to trigger when appropriate calls to `$assertoff`, as was done in the above example. Similarly if you have a long reset period where you want to turn off assertions for better simulation performance you can tie the related reset signal to control calls to `$assertoff()` to disable and `$asserton()` to reenable the assertions. An example in the case of a reset condition is below.

```
module assert_control ();
   :
   :
  initial begin
    if (foo.Reset == 1'b1)
      $assertoff (0, foo)
  end
endmodule

module foo ();
   :
   :
  property p1;
    @(posedge clk) disable iff (Reset) b ##1 c;
  endproperty
  assert property (p1);
endmodule
```

Benefit:

This will be a function of your testbench. We have an example in the past in AMD where assertions at some point accounted for 34% and that dropped to %18 with a switch to using `$assertoff` to disable assertions in one of the large IP blocks.

Tradeoff:

Whenever you create mechanisms to disable assertions, make sure the verif team has the assertions enabled when they are needed. Also, `$assertoff` is used to disable assertions across an entire module hierarchy level. If there are different enables used to control when the assertions are on/off in a module, you cannot use this method.

---

7. VCS option – "+vcs+learn+pli" – Limit your PLI access

Option:

PLI access levels control what signals are available for read/write, force and callback. If you can limit the PLI acc access of your testbench, VCS can better optimize the compile and improve runtime performance. VCS has however a few global options to enable PLI access that can easily be the default part of a testbench environment. Users should tend to avoid using global switches like `+acc+<number>` or defining PLI tab file acc entries that map to all "*" modules.

The one caveat here is that you might always be choosing to compile in access globally for supporting waveform dumping. In the case of Novas supported fsdb waveform dumping, the options `+vcsd` and `novas.tab` will enable some callback access across all modules. Even in this situation, while "callback" access might be enabled globally, you can still work to reduce the other acc access related to "read/write" and "force". At

AMD, we tend to always keep the access related to waveform dumping on all vcs compiles.

Use Model:

VCS supports a runtime option called "`+vcs+learn+pli`" to make VCS track and write out the modules accessed through PLI during the test run. It also tracks the type of acc access required. The output after simulation is complete is a tab file and this can be reapplied back into vcs compile through a `+applylearn+filename` to better optimize the compile. If this `+vcs+learn+pli` runtime option is run across a regression of tests and the resulting tab files are merged together, you can use this tab file list as a starting point for the limited acc access required to support your environment. There might be some iteration to add back acc access to modules required for some tests after this change is released, but that is simple to resolve. To see the benefits of using your new tab file, you need to ensure other options that enable global access are not still enabled. This includes avoiding switches like `+acc+<number>` and tab file entries in your environment that might apply to all modules – an example is provided below,

```
acc=rw,cbk:*
```

Benefit:

The below reflects running an AMD test with and without +applylearn option. Applying the +applylearn option requires the removal of +vcsd (which will disable waveform dumping support).

	Data (hr:min:sec)	Average (hr:min:sec)	Comments
Reference	Run 1 : 03:03:49 Run 2 : 03:04:14 Run 3 : 03:04:33	03:04:12 (1x)	7.66% gain with enabling +applylearn and removing +vcsd. waveform dumping.
+applylearn+<.tab> and removed +vcsd	Run 1 : 02:49:48 Run 2 : 02:50:27 Run 3 : 02:50:02	02:50:06 (0.9234x)	

Table 2 – Benchmark data for reducing PLI acc access

Tradeoff:

This method works well if you have areas where you build your design without waveform dumping support. After limiting your acc access if/when a user tries to PLI access a module that doesn't have access the simulation will simply fail with a ACC access error. In this case the user can simply add the PLI acc access for the module into the tab file that was originally created through the `+vcs+learn+pli` flow.

8. Environment Variable: NOVAS_FSDB_SKIP_CELL_INSTANCE

Option:

Fsdb waveform dumping is typically a big performance hit compared against a basic test run without dumping. If you are dumping all signals it's usually a performance hit of 2-3x compared to a no dump simulation. Fsdb dumping performance is also pretty linear. If you can reduce your dumping you can increase the performance of your fsdb runs. The same idea applies to the Siloti technology from Springsoft/Synopsys. The user can also make some smart decisions that can be easily implemented. Usually standard cells and memory macro models typically will already define themselves with a `` `celldefine/`endcelldefine `` verilog directives. If there are no `` `celldefine/`endcelldefine `` directives it can be easily added around these library cells. Most verification users don't need to debug signals inside of these models.

By setting "`setenv NOVAS_FSDB_SKIP_CELL_INSTANCE 1`" you can tell the fsdb dumper to skip dumping signals from any hierarchies defined as a cell. This helps to reduce the signals being dumped to fsdb which helps your simulation speed, and this is particularly helpful for gatesim simulation areas. You could also apply this to 3rd party IP areas where limited debug is needed or perhaps the code is already encrypted. Just add a `` `celldefine/`endcelldefine `` around the tops of hierarchies that can be removed from fsdb dumping.

Use Model:

Review your stdcell libraries and memory/macro models to confirm they are defined with `` `celldefine/`endcelldefine `` directives. The environment should set "`setenv NOVAS_FSDB_SKIP_CELL_INSTANCE 1`" as default. This can be easily changed by the end user and the test can be rerun without recompiling vcs. When users have the rare need to debug inside standard cells or memory/macro models, they can locally "`unsetenv NOVAS_FSDB_SKIP_CELL_INSTANCE`" and rerun their test.

Benefit:

Here are some benchmark data for an AMD RTL graphics testcase:

	Data (hr:min:sec)	Average (hr:min:sec)	Comments
Reference (no fsdb waveform dump)	Run1 : 02:59:36 Run 2 : 02:59:13 Run 3 : 02:59:14	02:59:21 (1x)	In the full fsdb waveform case, adding the NOVAS environment variable reduced the runtime for this testcase by 58.6%.
+ full fsdb	Run 1 : 12:20:05 Run 2 : 12:19:21 Run 3 : 12:18:34	12:19:20 (4.12x)	
+full fsdb + "setenv NOVAS_FSDB_... SKIP_CELL_... INSTANCE 1"	Run 1 : 05:06:09 Run 2 : 05:05:57 Run 3 : 05:05:55	05:06:00 (1.71x)	

Table 3 - Benchmark data for "setenv NOVAS_FSDB_SKIP_CELL_INSTANCE 1"

The results for the above a better than expected from earlier reviews in AMD. Also note that the results should be much better in gate simulation test cases.

Tradeoff:

For the odd case where a user needs visibility, they will have to rerun the simulation from start.

---

9. VCS option "-assert novpi"

Option:

VCS supports an option call "`-assert novpi`". For designs with assertions, if the user vpi code does not need to access information specific to assertions, you can do an optimization to remove some of the overhead due to vpi and the assertions. The VPI code AMD has does not need this extra data. One of our Synopsys contacts also noted that if that changes in the future, VCS would put out some kind of runtime message indicating that the VPI code cannot collect whatever information from the assertion that it needs.

Use Model:

This option can be applied for all simulations where coverage collection is not enabled. VCS will ignore the option if it is passed when assertion coverage is enabled and will present the following warning:

```
Warning-[CM_NOVPI] -assert novpi will be ignored
  Option '-assert novpi' cannot be used with flags that enable
coverage or debug on assertions e.g. '-cm assert', '-ova_cov',
'-assert dve', '-assert dump_sequences'.
```

Benefit:

Some benchmark results from an AMD graphics environment:

	Runtime (hr:min:sec)	Average (hr:min:sec)	Comments
Reference	Run1 : 03:02:26 Run 2 : 03:02:16 Run 3 : 03:01:42	03:02:08 (1x)	1.52% gain seen with adding –assert novpi option
add –assert novpi	Run 1 : 02:59:36 Run 2 : 02:59:13 Run 3 : 02:59:14	02:59:21 (0.985x)	

Table 4 - Benchmark data for "-assert novpi"

Tradeoff:

Should only be enabled for regressions where coverage data is not collected.

---

10. VCS "debug" options too expensive to enable by default

Option:

VCS has many user debug options that can be quite expensive in terms of runtime performance and should simply not be enabled on by default. This includes options for dumping SVA sequences, enabling access options to support interactive debug, and dumping multidimensional arrays. The data gathered below is just a few examples of the performance hit we see on a testcase at AMD.

Usage Model:

Users should be careful not to leave any of these VCS debug options on by default. The testbench environment should support user controls to enable the required debug options when needed.

Benefit:

The effect of these options will vary based on testcase, but the results below reflect applying these options for an AMD gfx testcase.

Option:	Benchmark Results (hr:min:sec)	Cost
Waveform Dumping value changes of system verilog multi-dimentional arrays to fsdb file This involves adding runtime option `+fsdb+mda=on` or "`setenv NOVAS_FSDB_MDA 1`" or `+mda` if part of input file to `$fsdbDumpvarsByFile`[1]	Reference run with full fsdb: Run 1: 05:06:09 Run 2: 05:05:57 Run 3: 05:05:55 Avg = 05:06:00 (1x) Full fsdb + "`+mda`" option: Run 1: 05:21:13 Run 2: 05:21:39 Run 3: 05:21:37 Avg = 05:21:30 (1.051x)	5.06% increase
Waveform Dumping for SVA assertions results to the fsdb file This involves adding a call to `$fsdbDumpSVA` as part of your waveform dumping calls. [1]	Reference run with full fsdb: Run 1: 05:06:09 Run 2: 05:05:57 Run 3: 05:05:55 Avg = 05:06:00 (1x) Full fsdb + `$fsdbDumpSVA` Run 1: 05:09:13 Run 2: 05:08:14 Run 3: 05:08:48 Avg = 05:08:45 (1.01x)	0.88% increase in runtime This is quite a bit smaller than expected for the testcase. This has at some point been a very expensive option in AMD.
Interactive Debug Options for UCLI and DVE VCS compile options: `-debug_pp`	Reference run (no fsdb) Run 1: 02:59:36 Run 2: 02:59:13 Run 3: 02:59:14 Avg = 02:59:21 (1x)	15.50% increase in runtime for adding `-debug_pp` `-debug_pp` is the

-debug -debug_all Details : Each option above would be use exclusively, and enable differing levels of debug access for UCLI and DVE user debug. [1]	 Add -debug_pp (no fsdb) Run 1: 03:27:19 Run 2: 03:27:19 Run 3: 03:26:49 Avg = 03:27:09 (1.155x)	least expensive of the 3 options. -debug and -debug_all would have an even greater performance degradation.
Waveform Dumping for Assertion Sequencies VCS compile options: -assert enable_diag -assert vpiSeqBeginTime -assert vpiSeqFall Details : enable_diag Enables further control of results reporting with runtime options vpiSeqBeginTime Enables you to see the simulation time that a SystemVerilog assertion sequence starts when using Debussy vpiSeqFail Enables you to see the simulation time that a SystemVerilog assertion sequence doesn't match when using Debussy [2]	Reference run (no fsdb) Run 1: 03:03:49 Run 2: 03:04:14 Run 3: 03:04:33 Avg = 03:04:12 (1x) Add -assert enable_diag -assert vpiSeqBeginTime -assert vpiSeqFall Run 1: 03:29:29 Run 2: 03:26:30 Run 3: 03:26:21 Avg = 03:27:27 (1.124x)	12.40 % increase in runtime when enabling -assert enable_diag -assert vpiSeqBeginTime -assert vpiSeqFall for the situation where no waveform being dumped.

Table 5 - Performance effects of a few VCS 'debug' options

Tradeoff:

Users who need to have access to these detailed debug options will need to rerun with the relevant debug option enabled.

11. Summary

This paper has given a quick summary of a strategy for running a proper simulation benchmark and a description of a few legacy areas to review for improving SOC VCS simulation performance. These have provided some benefit for users in AMD. It is the hope of the author that you can apply some of these ideas for your own testbench.

Here is a checklist summary table to help point you to simulation performance optimizations:

Do you use "this" in your environment?	Go check out these options
Run simulation benchmarks to gauge your performance	Go see "A Benchmarking Strategy Recommendation"
Not currently running time profiles	Go See "Time Profiling"
Use SVA assertions	Go see "$assertoff" and VCS "debug" options
Run some tests without coverage collection?	Go see "VCS option -assert disable_cover" and "-assert novpi"
FSDB Waveform Dumping	Go see "setenv NOVAS_FSDB_SKIP_CELL_INSTANCE 1" and VCS "debug" options
Not currently using vcs option –top	Go see "VCS option –top"
Use PLI code	Go see "VCS option +vcs+learn+pli"
Don't regularly review for performance issues	Go see list of areas presented in "introduction". What was described here in this paper was limited to VCS options which isn't the only option or necessarily the best area to optimize. Go through all the items listed in the introduction as associated to your own design environment.

Table 6 - Summary Checklist

To end the paper, here are the author's top three recommendations from the topics discussed:

1. Review the options defined in the last section describing debug options. If any are present as a default in your environment, clarify the cost and consider removing.
2. Create a VCS time profile and review the report carefully.
3. These are all easy to deploy. Add a "`setenv NOVAS_FSDB_SKIP_CELL_INSTANCE 1`" as a default in your environment. For simulations run without coverage collection, review and release the options "`-assert disable_cover`" and "`-assert novpi`".

12. REFERENCES

[1] 2012.09-SP1-1 VCS Online Documentation
[2] VCS/VCSi™ LCA Features G-201.09-SP1-1, April 2013
[3] 2013.04 Verdi³ and Siloti Command Reference Manual