Optimization techniques to improve simulation memory performance

Aditya Musunuri – Freescale
Amol Bhinge - Freescale
Narayana Koduri - Synopsys

Freescale Semiconductor Inc.
Austin TX, USA

www.freescale.com

ABSTRACT

Complexity of SoC designs is growing phenomenally, this increases the demand for more verification resources significantly. With multiple chips are being taped-out each year based on time-to-market requirements, costs of verification resources need to be reduced and optimized for better profitability. Machine computing resources are one of the key expense contributing to the overall verification cost. Optimizing and effectively managing the machine resources helps not only with verification cost but also with project schedules.

This paper describes the methodology and best practices applied by Freescale in collaboration with Synopsys to achieve optimum regression runtime memory performance for RTL simulations. The methodology deploys the newly available run-time memory profiler as an advanced debugging technique to identify performance bottlenecks and to enable corrective actions.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

As the complexity of SoC Designs is increasing, the complexity of its verification is increasing multi-fold. This includes a more complex testbench environment and an increased number of testcases. The complexity of the SoC and its associated verification has a direct impact on the performance of the simulation tool. Two performance metrics are of interest here: the time to perform the task, whether it is compilation or simulation, and the memory utilization while doing it. An increase in any of these directly corresponds to an increase in the computing resources needed for executing a project. However, in reality, it is not possible to increase the allocation of computing resources for every project. The alternative is to benefit from the many enhancements and optimizations that have been implemented by the simulator tools to tackle and improve these performance metrics.

In this paper, we focus on the simulation time memory utilization. During the execution of one of our project, we noticed a shortage of LSF resources, with the requested amount of memory, which increased the turn-around time and was slowing us down. After investigating into the LSF shortage, we found out that each VCS simulation job was consuming higher memory than we expected based on our previous project. We then, with the help of Synopsys® team, analysed the causes for such memory requirement and enabled corrective actions to bring it down to permissible levels.

2. Design

The QorIQ LS2 family of communications processors delivers unprecedented performance and integration for the smarter, more capable networks of tomorrow. Shown in Figure 2.1, the QorIQ LS2085A multicore processor combines eight ARM® Cortex®-A57 cores with the advanced, high-performance datapath and network peripheral interfaces required for networking, telecom/datacom, wireless infrastructure, military and aerospace applications. The QorIQ LS2 family includes Freescale's second generation data path acceleration architecture (DPAA2) which provides the infrastructure required to support simplified and secure networking interface and accelerator sharing by multiple general purpose CPU cores.[1]

QorIQ LS2 family processors integrate up to eight 10 Gb/s and eight 1 Gb/s Ethernet interfaces with L2 switching capability for PCIe® controllers (supporting SR-IOV) and next-generation SATAIII and USB3 controllers. The QorIQ LS2 processors contain an Advanced IO Processor that offloads the general purpose ARM Cortex®-A57 cores with in-line or fully autonomous networking functions, involving complex look-ups, header manipulations, and even encapsulations and encryption.[1]

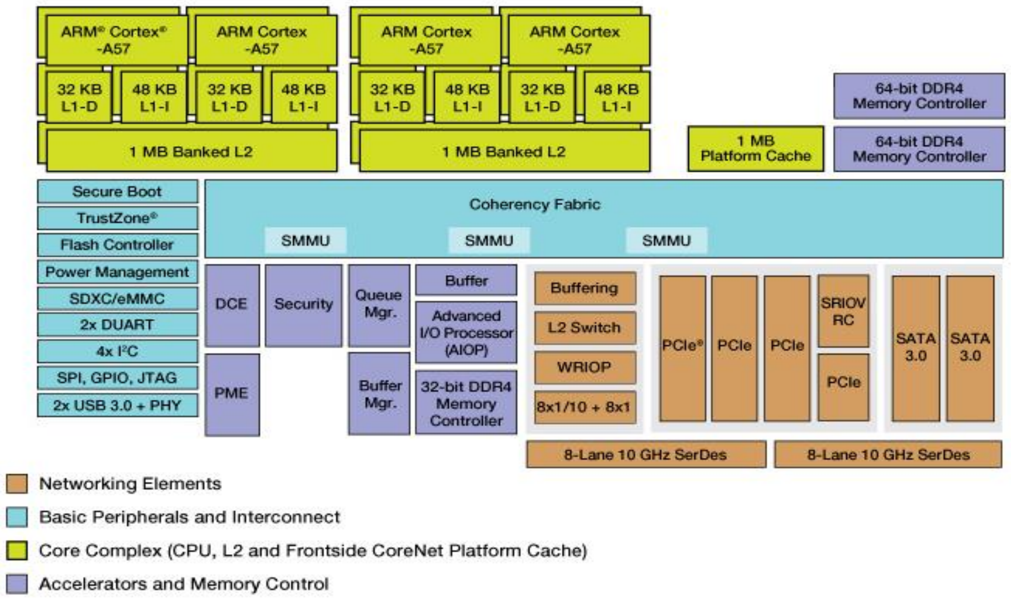


Figure 2.1 QorIQ LS2085A Communication Processor

3. Testbench Environment

This section describes the testbench environment.

Methodology

The testbench is centered on a UVM 1.1 based chassis. All elements in the verification environment must be synchronized to this central fabric which holds sequences that will perform the main device functionality and test flow. The testbench includes elements written in various methodologies and languages such as: Verilog, System Verilog, UVM, C++, ROCOO (Freescale Internal - Random On Chip Object Oriented Language).

Testbench Architecture

As shown in Figure 3.1, there are two key components in the testbench architecture i.e., module and program.

Module part

The testbench top module includes the following main elements apart from the DUT:

- DUT_IO_CONNECTION, DUT_SERDES_CONNECTION – A MUX for the IO/SERDES pins in the DUT.
- <env_name>_TB_TOB - Each env will include sub envs with similar characteristic, for example: mem_tb_top, perif_tb_top, etc.

Program part

The elements in the program part of the testbench are both UVM classes and non UVM classes. The main elements of the program part are:

- SOC_TEST – Executes a Virtual Sequence on Virtual Sequencer for each phase.
- SOC_ENV – wrapper for all the various envs. Also includes SOC related classes.
- <env_name>_ENV – a specific env for each block/driver/VIP.

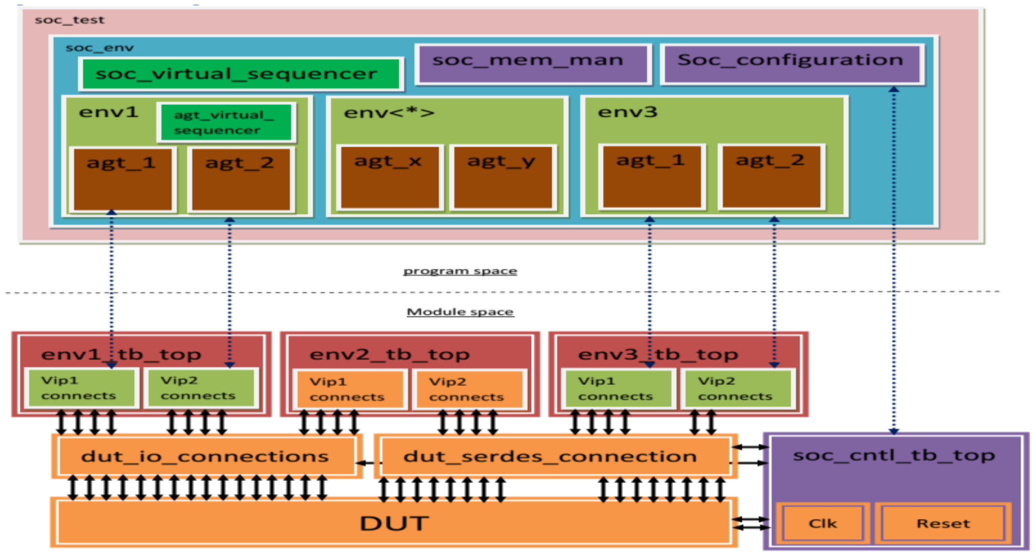
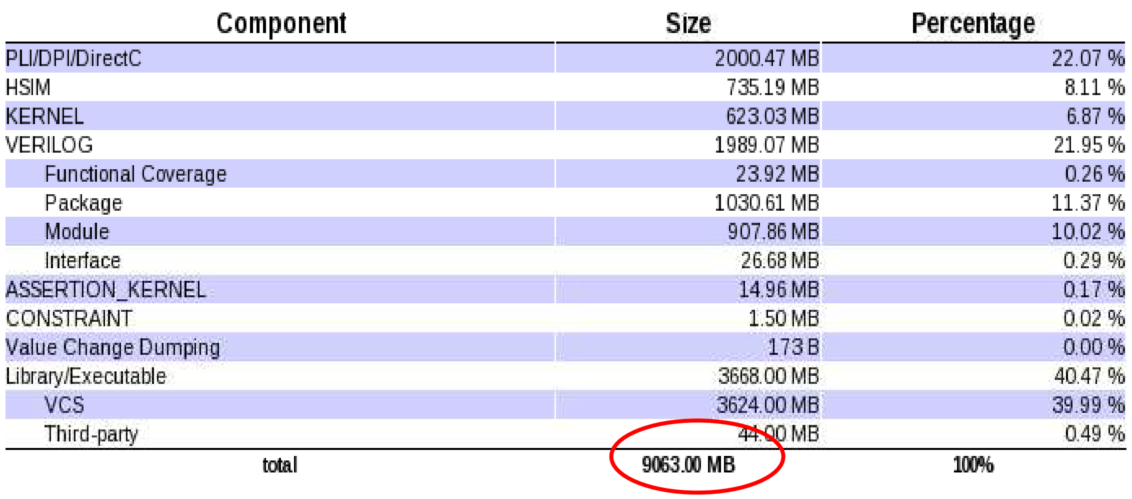


Figure 3.1 SoC Testbench Block Diagram

4. Findings and Results

To manage the overall verification cost, we do our best to use our machine resources efficiently. Several years back, by collaborating with Synopsys, we created a setup to have several slots that will only need 6GB memory using our LSF system. However, with LS2 design, we noticed that memory consumption is 9.1GB. So, our goal was to bring the memory footprint in the range of 6GB to meet our LSF setup requirements. We started studying memory consumption using VCS Simprofiler. VCS Simprofiler feature has been enhanced to show the details of simulation memory profiler in addition to simulation runtime profile. Enabling this feature helps generating various memory profiles based on Modules, Constructs, Instances, PLI/DPI/DirectC, Constraints Solver, Functional Coverage, Dynamic Memory, etc.

Simprofiler can be viewed either with html or text based. Figure 4.1 shows the HTML based Simprofiler report generated when we started investigating our base setup.



Component	Size	Percentage
PLI/DPI/DirectC	2000.47 MB	22.07 %
HSIM	735.19 MB	8.11 %
KERNEL	623.03 MB	6.87 %
VERILOG	1989.07 MB	21.95 %
Functional Coverage	23.92 MB	0.26 %
Package	1030.61 MB	11.37 %
Module	907.86 MB	10.02 %
Interface	26.68 MB	0.29 %
ASSERTION_KERNEL	14.96 MB	0.17 %
CONSTRAINT	1.50 MB	0.02 %
Value Change Dumping	173 B	0.00 %
Library/Executable	3668.00 MB	40.47 %
VCS	3624.00 MB	39.99 %
Third-party	44.00 MB	0.49 %
total	9063.00 MB	100%

Figure 4.1 Simprofiler with default setup

The profiler report shows that the key contributors are following:

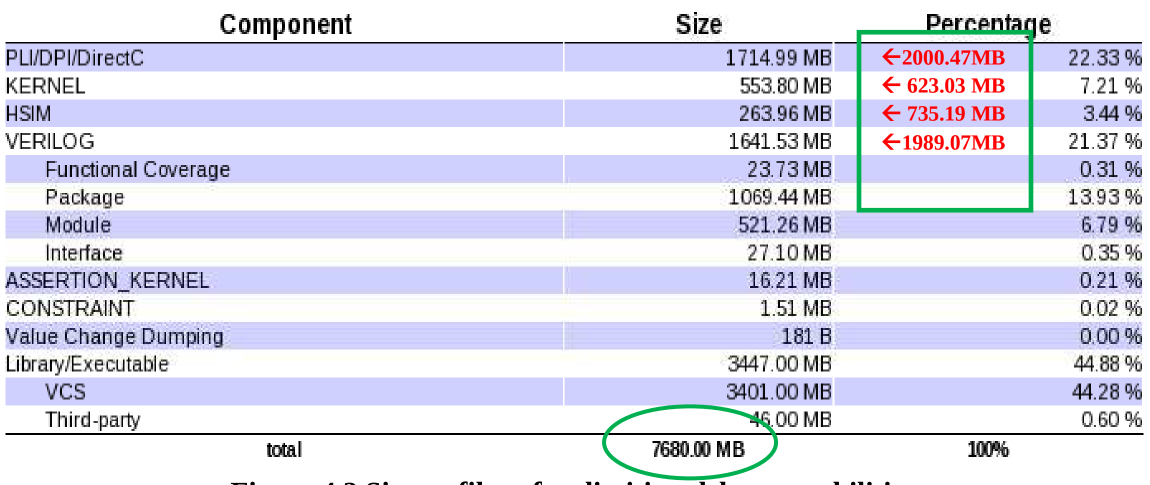
a. PLI/DPI/DirectC – 2000.47 MB
b. Verilog (Module, Package) – 1989.07 MB
c. Library Executable – 3668.00 MB

Let us take a look at each of these contributors in more details.

a. PLI/DPI/DirectC:

The memory consumed in this category is primarily due to debug capabilities enabled with PLI tab, DPI and DirectC Interfaces. We knew that our setup doesn't have major DPI/DirectC interfaces. So, we focused on PLI based debug capabilities. After looking at the setup, we found that we are using VCS compile option `–debug` which enables debug visibility as well control. This debug option provides several debug access capabilities such as forcing, writing or depositing nets, registers, etc. It enables also setting value and time breakpoints. Enabling a specific debug capability means, we are restricting VCS from performing any possible optimization. This in effect causes VCS compiler to generate bigger code and therefore creates overhead on simulation memory utilization and runtime. But, these debug capabilities are redundant for performing regression which has 7000 testcases. So, we replaced with the `–debug` option with `–debug_access+r` which only allows read capabilities.

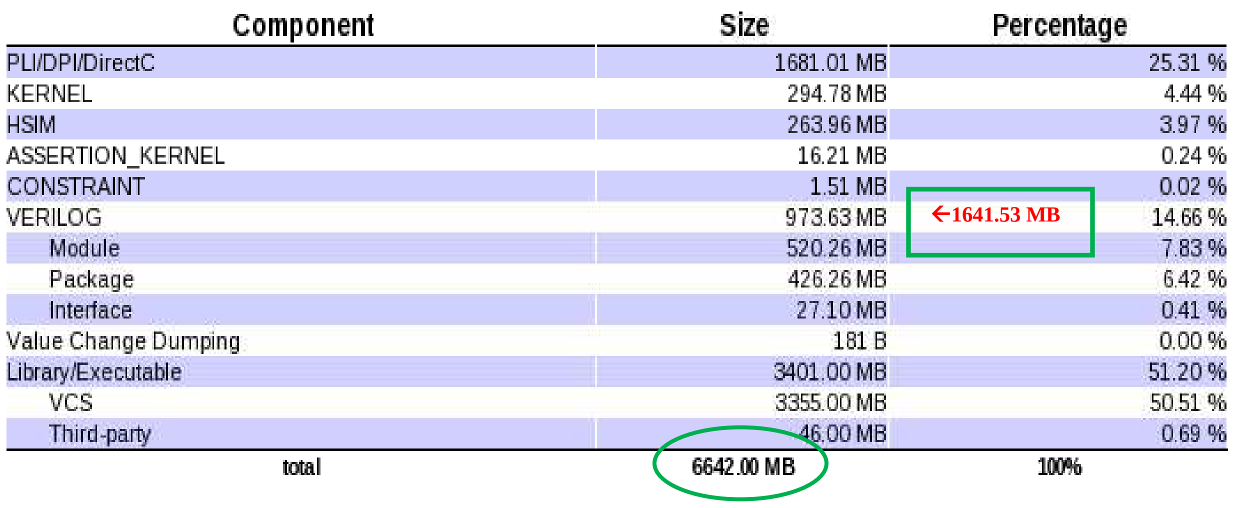
At the same time, as a SoC team, we were not sure what debug capabilities are needed to selectively choose them. However, VCS has PLI learn capability which, when enabled, it can learn what debug capabilities are needed during simulation. So, we leveraged this feature and generated a *tab* file, called learn_pli.tab from a specific simulation test. VCS generates pli_learn.tab which can be specified with `+applylearn+<learn_pli.tab>` while compiling the design. This was in addition to the option "`–debug_access+r`" mentioned earlier. So, we changed our setup accordingly and noticed that total memory consumption improved by 15%, down to 7680MB from 9063MB. As you can see in the highlighted green box in Figure 4.2, improvement is across different components.



Component	Size		Percentage
PLI/DPI/DirectC	1714.99 MB	←2000.47MB	22.33 %
KERNEL	553.80 MB	← 623.03 MB	7.21 %
HSIM	263.96 MB	← 735.19 MB	3.44 %
VERILOG	1641.53 MB	←1989.07MB	21.37 %
Functional Coverage	23.73 MB		0.31 %
Package	1069.44 MB		13.93 %
Module	521.26 MB		6.79 %
Interface	27.10 MB		0.35 %
ASSERTION_KERNEL	16.21 MB		0.21 %
CONSTRAINT	1.51 MB		0.02 %
Value Change Dumping	181 B		0.00 %
Library/Executable	3447.00 MB		44.88 %
VCS	3401.00 MB		44.28 %
Third-party	46.00 MB		0.60 %
total	7680.00 MB		100%

Figure 4.2 Simprofiler after limiting debug capabilities

b. Verilog (Module, Package):
This component shows memory consumed due to modules, packages, interfaces and functional coverage. In Figure 4.2, you can notice that, Verilog remains as one of the major contributors. We studied to find what specific package is taking more memory. As part of our investigation, we found that some of our VIPs are enabled with transaction coverage which is leading to higher memory consumption. However, memory consumption is shown under Package where as it should be shown under Functional Coverage. Again, this transaction coverage data may be useful during initial debug to understand the coverage info and not needed for regression setup. So, we disabled the coverage by adding a VCS option "`-covg_disable_cg`". This helped us improving the total memory consumption by another 13%, down to 6642MB from 7680MB, as shown in Figure 4.3.



Component	Size	Percentage
PLI/DPI/DirectC	1681.01 MB	25.31 %
KERNEL	294.78 MB	4.44 %
HSIM	263.96 MB	3.97 %
ASSERTION_KERNEL	16.21 MB	0.24 %
CONSTRAINT	1.51 MB	0.02 %
VERILOG	973.63 MB	14.66 %
Module	520.26 MB	7.83 %
Package	426.26 MB	6.42 %
Interface	27.10 MB	0.41 %
Value Change Dumping	181 B	0.00 %
Library/Executable	3401.00 MB	51.20 %
VCS	3355.00 MB	50.51 %
Third-party	46.00 MB	0.69 %
total	6642.00 MB	100%

Figure 4.3 Simprofiler after disabling coverage data

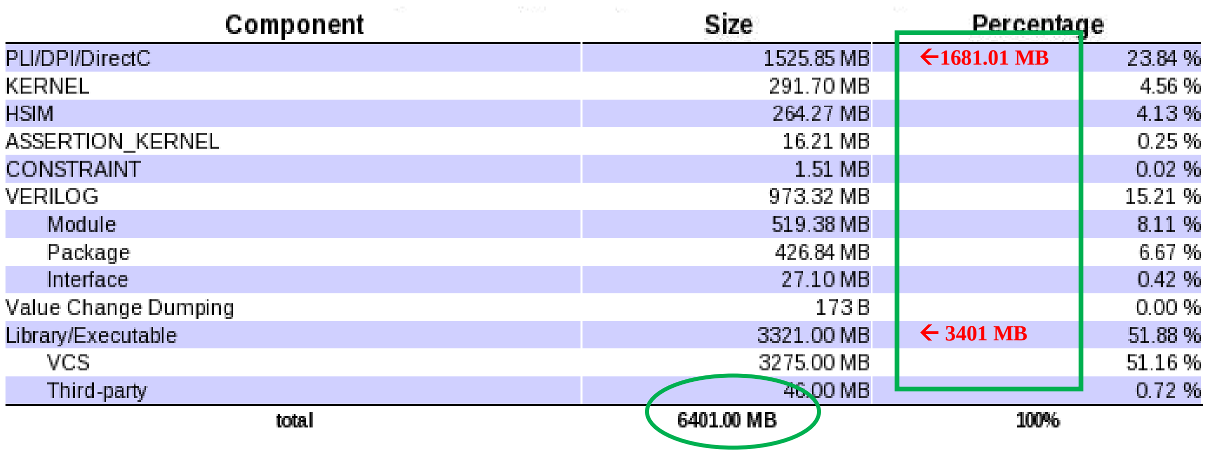
c. Library Executable
Finally, we focused on this component which is related to `vcs` executable and any 3rd party programs part of the executable. After further investigation with Synopsys team, we found that this is an overhead caused by a message logger of some specific VIPs in the testbench environment. So, we replaced `+vpi` VCS option with `+vpi+1` option to eliminate the memory overhead. This option limits the behavioural information at compile-time, but preserves the structural information.

Also, while working with Synopsys team, we found that following options are kind of legacy and not required, at the same time may lead to additional overhead. So, we removed those options and refreshed the compile scripts accordingly:

a. Remove `+memcbk`
b. Remove `+vcsd`
c. Remove `${NOVAS_HOME}/share/PLI/VCS/LINUXAMD64/pli.a`
```
-P ${NOVAS_HOME}/share/PLI/VCS/LINUXAMD64/novas.tab
```

Note that the options in c. above are related to enable dumping FSDB for Verdi. The removal of these options doesn't take the ability to dump FSDB because using the option "`-debug_access`" enables the same by default.

This helped us bringing overall memory footprint to 6GB which is our original target, as shown in Figure 4.4.



Component	Size		Percentage
PLI/DPI/DirectC	1525.85 MB	←1681.01 MB	23.84 %
KERNEL	291.70 MB		4.56 %
HSIM	264.27 MB		4.13 %
ASSERTION_KERNEL	16.21 MB		0.25 %
CONSTRAINT	1.51 MB		0.02 %
VERILOG	973.32 MB		15.21 %
Module	519.38 MB		8.11 %
Package	426.84 MB		6.67 %
Interface	27.10 MB		0.42 %
Value Change Dumping	173 B		0.00 %
Library/Executable	3321.00 MB	← 3401 MB	51.88 %
VCS	3275.00 MB		51.16 %
Third-party	46.00 MB		0.72 %
total	6401.00 MB		100%

Figure 4.4 Simprofiler with regression setup

Note that the Simprofiler feature itself has some overhead when enabled; hence the report is showing 6.4GB. This is because Simprofiler generates lot of data to track the memory consumption across different components. Hence, it consumes additional memory, so this option should not be used by default. However, it can be enabled for debugging purposes.

Table 4.1 summarizes the performance observed with original and updated setup.

VCS 2014.12-SP1	Base Numbers	Current Numbers	Progress
Setup	Original	Regression setup	
Runtime	806 secs	583 secs	28%
Runtime Memory	9 GB	6.0 GB	33%

Table 4.1 Results (Dump Disabled) with original and regression setup

5. Findings and Results with "DUMP"

After we get control on memory performance on regression setup, we also looked at memory performance with waveform dumping, for both VPD and FSDB. The default dumping in our environment was VPD. We established the base line based on that. Then upon Synopsys recommendation, we tried FSDB dumping since it has more flexibility and control over what to dump, and for its advanced debug capabilities. With the regression setup, we observed noticeable improvement in runtime memory and also in the FSDB dump file size. But, simulation runtime was almost same. However, starting with VCS 2014.12, there is a way to dump FSDB natively instead of dumping through PLI. Synopsys recommended to enable this by specifying env variable "`FSDB_DUMPER_VDI`" before performing simulation. This helped us achieving significant runtime improvement.

Table 5.1 shows the summary of numbers.

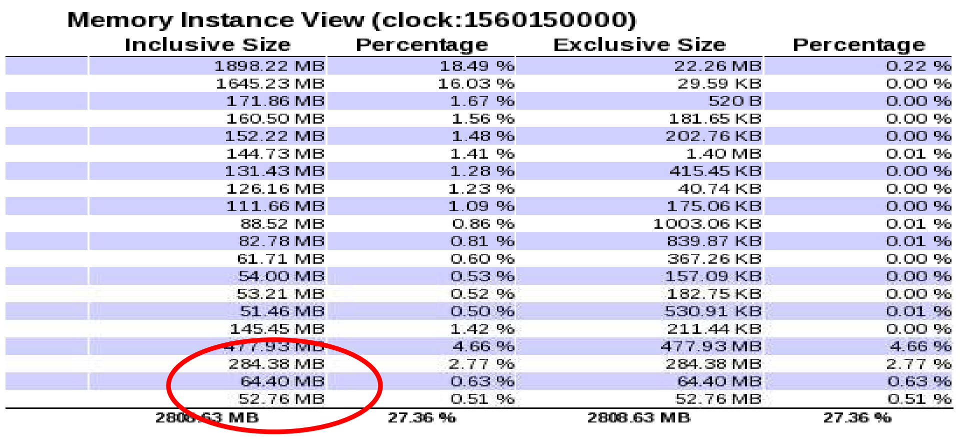
Ver1 – Optimized Setup
`With all options discussed in section 4`

Ver2 – Native FSDB Dumping
`Ver1 + environment variable FSDB_DUMPER_VDI set to 1`

VCS 2014.12-SP1 Verdi 2014.12-SP1	Base (vpd)	Ver1 (fsdb)	Ver2 (fsdb)	Progress
Setup	**Regression Setup**		**Regression Setup+Native**	
Runtime (secs)	**1469**	**14080**	**1005**	**46%**
Runtime Memory (GB)	**13.2**	**10**	**9.6**	**37%**
Dump Size (MB)	**149**	**135**	**120**	**24%**

Table 5.1 Results (Dump enabled) with original and updated setup

We investigated further to see if we can improve the performance. From VCS Simprofiler, we focused on "Module" component as it is the major contributor. In Figure 5.1, you can see that highlighted instances(instance refererecces removed for confidentiality reasons) are taking some memory.



Memory Instance View (clock:1560150000)

	Inclusive Size	Percentage	Exclusive Size	Percentage
	1898.22 MB	18.49 %	22.26 MB	0.22 %
	1645.23 MB	16.03 %	29.59 KB	0.00 %
	171.86 MB	1.67 %	520 B	0.00 %
	160.50 MB	1.56 %	181.65 KB	0.00 %
	152.22 MB	1.48 %	202.76 KB	0.00 %
	144.73 MB	1.41 %	1.40 MB	0.01 %
	131.43 MB	1.28 %	415.45 KB	0.00 %
	126.16 MB	1.23 %	40.74 KB	0.00 %
	111.66 MB	1.09 %	175.06 KB	0.00 %
	88.52 MB	0.86 %	1003.06 KB	0.01 %
	82.78 MB	0.81 %	839.87 KB	0.01 %
	61.71 MB	0.60 %	367.26 KB	0.00 %
	54.00 MB	0.53 %	157.09 KB	0.00 %
	53.21 MB	0.52 %	182.75 KB	0.00 %
	51.46 MB	0.50 %	530.91 KB	0.01 %
	145.45 MB	1.42 %	211.44 KB	0.00 %
	477.93 MB	4.66 %	477.93 MB	4.66 %
	284.38 MB	2.77 %	284.38 MB	2.77 %
	64.40 MB	0.63 %	64.40 MB	0.63 %
	52.76 MB	0.51 %	52.76 MB	0.51 %
	2808.63 MB	27.36 %	2808.63 MB	27.36 %

Figure 5.1 Simprofiler (Dump Enabled) Instance View

During FSDB dump, user has an ability to control the information that is being dumped. For example, if testbench and uvm information is not necessary for your debug, user can avoid the dump for the same. In a similar manner, user can also limit the dump to registers only. We did couple of experiments, to see the impact of these examples. Here is the way to do this using VCS Unified Command Line Interface (UCLI):

Exp1 - Limit dumping to DUT excluding 'testbench'
```
fsdbDumpvars 0 "testbench.top‟
```

Exp2 - Limit dumping to DUT excluding 'testbench' and dump only Registers
```
fsdbDumpvars 0 "testbench.top‟ ‟+Only_Reg”
```

The results of these experiments are published in **Table 5.2 Results with Dump Experiments**, and they show that there has been significant improvement in the memory foot print, dump size and some improvement in run time.

VCS 2014.12-SP1 Verdi 2014.12-SP1	Ref (fsdb)	Exp1 (fsdb)	Exp2 (fsdb)	Progress
Setup	Native	+Limit to DUT	+Reg Only	
Runtime (secs)	1005	1115	983	3 %
Runtime Memory Inst View (GB)	2.8	2.3	1.6	43%
Runtime Memory (GB)	9.6	9.2	8.1	16 %
Dump Size (MB)	120	65	15	8 X

Table 5.2 Results with Dump Experiments

In the similar fashion, there are several other ways to control the information that is being dumped. Please refer to Verdi documentation[3] for more details on these commands.

6. Conclusions

Using VCS Simprofiler, we were able to investigate runtime memory consumed across different components of VCS. As shown in Table 4.1, this exercise not only helped improving the memory overhead, but also helped in improving the runtime significantly. We found that it is very important to review debug access capabilities used and make sure they are appropriate for the task, and if necessary use VCS apply learn flow to learn the debug capabilities required and limit to those capabilities.

Similarly, as discussed in DUMP section, based on debug requirements, user can disable specific data using different FSDB commands to optimize memory, dump size and run time further. As discussed in Table 5.1 and Table 5.2, with recommended setup and couple of experiments we have seen phenomenal performance improvement. So, it is highly recommended to study your debug methodology and dump only necessary information for best performance, runtime and dump size.

Recommendations:

a. Run the profiler periodically as it is a good way to keep an eye on the memory footprint.
b. Between projects, review compile and runtime options to make sure they are up to date.
c. Work with Synopsys application engineers to learn about the latest the simulation tool has to offer in new versions, especially for performance related options.

Table 6.1 summarizes all the setup recommendations discussed in this paper.

Phase	Option Name	Action
Compile Time	-debug_access+r	Use instead of –debug
	-P learn_pli.tab	Add the generated learn_pli.tab
	-covg_disable_cg	Add to disable cover group collection
	NOVAS tab file	Remove this option (include with –debug_access)
	+vpi+1	Use instead of +vpi
	+vcsd, +memcbk	Remove these options
Run time	setenv FSDB_DUMPER_VDI 1	Add this env variable (will be default in future Verdi versions)

Table 6.1 Recommended Compile and Runtime setup

Future Enhancements:

a. Limit the memory overhead with Simprofiler
b. Categorize the memory overhead caused by transaction coverage under coverage instead of showing it under package
c. Ability to redirect databases similar to coverage databases and to generate summary reports for a given regression run

Next steps:

a. The above analysis and optimization was done for simulation runs in regression. Similar analysis needs to be performed for coverage related simulations also.
b. Evaluate +rad feature for further performance optimization.

7. References

[1] QorIQ® LS2085A 64-bit Multicore Communications Processors
http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=LS2085A

[2] VCS 2014.12 Documentation

[3] Verdi 2014.12 Documentation